REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEINEMENT SUPERIEUR
ET DE LA RECHERCHE SCIENTIFIQUE
UNIVERSITE DE BATNA
FACULTE DES SCIENCES DE L'INGENIEUR

MEMOIRE

Présenté au

DEPARTEMENT D'ELECTRONIQUE

Pour l'obtention du diplôme de

MAGISTER EN MICROELECTRONIQUE

Option : IC Design

Par

Badraddine BEZZA

Ingénieur, Département d'Electronique-Université de Batna

Intitulé

Micro processeur a instructions variables

Devant le jury constitué de :

Dr. S.AOUAGHLENT	M.C.	U. Batna	Président
Dr. Z.DIBI	M.C.	U. Batna	Rapporteur
Dr. A.CHAABI	Pr.	U. Constantine	Examinateur
Dr. N.ATHEMENA	M.C.	U. Batna	Examinateur

2010

DEDICACES

Je dédie ce travail à :

Ma mère,

Mon père,

Ma femme,

Mes frères et mes sœurs,

Tous mes amis sans exception.

Remerciements

Je tiens à remercier, en premier lieu, **Mr DIBI Zohir**, chef de département d'Electronique et maître de conférences à l'université de Batna, pour avoir accepté d'encadrer ce travail. Ses conseils judicieux, ses suggestions pertinentes et la chaleur de ses encouragements m'ont permis de mener ce projet à terme.

Je remercie aussi les membres du jury pour avoir pris sur leur temps et sur leurs occupations multiples, ces moments de lecture et de réflexion en vue de cette soutenance.

A tous mes collègues et amis, je dis encore merci pour votre sollicitude et votre compréhension.

A ma famille, à mes parents, à mes proches, et à tous les miens, merci à tous, pour tout.

Symbole	Description
ASIC	Application Specific Integrated Circuit
BRAM	Block RAM
CISC	Complex Instruction Set Computer
CLB	Configurable Logic Block
EEPROM	Electrically-Erasable Programmable Read-Only Memory
FSM	Final State Machine
FPGA	Field Programmable Gate Array
ISE	Integrated Software Environment
ISA	Instruction Set Architecture
JTAG	Joint Test Action Group
LC	Logic Element
LUT	Lock Up Table
MGA	Masked Gate Array
MUX	Multiplexer
PCB	Printed Circuit Board
SR	Shift Register
SRAM	Static RAM
RISC	Reduced Instruction Set Computer
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
HDL	Hardware Description Language

Sommaire

Introduction générale

Introduction générale

Depuis la création du premier circuit intégré, la principale force motrice pour l'industrie des semi-conducteurs a été entièrement le processus de mise à l'échelle (scaling). Ce processus consiste en la réduction de la taille physique des transistors et des fils d'interconnexion, ce qui permet de placer plus de dispositifs sur chaque puce et d'implémenter plus de fonctions complexes. Malgré la complexité croissante des circuits intégrés, la fonction d'un circuit intégré numérique est d'exécuter un algorithme.

Il existe deux principales méthodes du calcul traditionnel que sont utilisé pour l'implémentation des algorithmes [1]. La première consiste à utiliser le hardware (ou un ASIC) pour exécuter les opérations. Ces ASIC sont conçus spécifiquement pour exécuter un calcul donné, ils peuvent être très efficaces, avec une utilisation réduite des ressources et moins de consommation d'énergie par rapport à d'autres implémentations [2] [3]. Toutefois, après sa fabrication, le circuit ne peut pas être modifié. Les processeurs sont une solution beaucoup plus flexible, ils exécutent un ensemble d'instructions pour effectuer un calcul. En changeant les instructions au niveau du software, les fonctionnalités du système se voient modifiées et cela sans toucher au hardware. Cependant, l'inconvénient de cette flexibilité est que le rendement en souffre, et est bien inférieur à celui d'un ASIC. Le calcul Reconfigurable est destiné à combler l'écart entre le matériel et le logiciel, en réalisant des performances potentiellement beaucoup plus élevés que les logiciels, tout en maintenant un niveau plus élevé de flexibilité que le matériel [4].

Les activités de recherche dans le domaine des ASIP (*Application Specific Instruction Processors*) ont prouvé que l'utilisation des ressources complexes des chemins de données au lieu de primitives (comme additionneur et multiplicateurs) améliore les performances de telles applications [4], [5], [6]. Les parties critiques de l'application, appelées noyaux, sont déplacées à partir du logiciel au matériel reconfigurable comme FPGA [7], [8].

Dans les systèmes qui utilisent un processeur en conjonction avec du matériel reconfigurable, il existe quatre différentes architectures conçues pour être utilisés dans l'implémentation de ces noyaux [1], [8]. La variation primaire entre ces architectures est le degré de couplage avec le processeur principal. En premier, les noyaux peuvent exécuter les instructions en tant qu'unités fonctionnelles sur le chemin de données du processeur principal, ceci est le plus étroitement couplé au processeur et implique l'utilisation d'instructions personnalisées [9]. Deuxièmement, les noyaux peuvent être utilisés comme un coprocesseur. Ils

peuvent être utilisés comme en troisièment un processeur indépendant dans un environnement multiprocesseur. Enfin, sous la forme la plus faiblement couplée comme une unité de traitement autonome externe. Ce modèle est similaire à celui des postes de travail en réseau, où le traitement peut se produire pour de très longues périodes de temps avec peu de communication.

Dans les systèmes qui utilisent des instructions personnalisées (exécutent les noyaux dans les unités fonctionnelles), il a été prouvé qu'un jeu d'instructions bien conçu peut réduire les temps d'accès registre et mémoire, et donc la consommation d'énergie globale du système [5]. Autre avantage des instructions personnalisées est qu'elles peuvent éliminer l'utilisation d'instructions multiples, réduisant ainsi les coûts de recherche/décodage et améliorant la taille du code [10]

Les instructions personnalisées ne nécessitent pas d'être fixée à temps. Au lieu d'avoir toute instruction personnalisée (fonction) utilisée ou non, il serait judicieux de coder (placer ou reconfigurer) les instructions active à un moment donné directement dans le matériel ce qui permet de réduire le coût du matériel et la consommation d'énergie. La vitesse pourrait également être augmentée du fait que plusieurs unités matérielles spécialisées peuvent être implémentées plutôt que d'avoir une implémentation matérielle générale [7]. Toutefois, le temps long de reconfiguration limite l'applicabilité de la reconfiguration à l'exécution(en temps réel). Avec des dispositifs de plus en plus grands et plus rapides, il est maintenant possible de concevoir des systèmes reconfigurables à l'exécution sans avoir à reconfigurer l'ensemble du dispositif [11] [12] [13] [14], c'est ce qu'on appelle la reconfiguration partielle.

Pour utiliser efficacement la logique reconfigurable, les instructions personnalisées doivent être identifiés. Le concepteur doit exécuter le profiling et l'analyse en boucle du code séquentiel (l'algorithme) comme une première étape d'optimisation [8], puis les instructions personnalisées sont conçues en utilisant la méthodologie RTL (Register Transfer Level) ou autres méthodologie de conception matériels. Enfin, le code original doit être modifié pour utiliser le matériel (instructions personnalisées). Ces mesures sont appelées méthode de partitionnement logiciel/matériel. Dans les techniques de partitionnement logiciel/matériel composées d'un processeur et d'un FPGA, l'unité FPGA est considérée comme une extension du processeur [7]. Une conception en utilisant la méthodologie RTL pour la plate-forme FPGA donne des avantages de haute performance en terme de vitesse d'exécution, et une conception logiciel pour un processeur donne les avantages de moins d'effort de conception et moins de temps de

vérification, la combinaison du processeur et FPGA donne les avantages des deux plates-formes. Il est important de noter que si les efforts de conception sont trop élevés ou l'implémentation RTL dépasse la capacité du FPGA, l'utilisation de la plate-forme combinée est obligatoire.

Dans le monde des FPGA, il existe deux types de cœur processeur; cœur processeur soft et cœur processeur hard. Les cœurs de processeur soft contrairement aux cœurs processeur hard sont conçus avec un langage HDL et implémentés en utilisant les ressources logiques d'une FPGA. Un avantage d'utiliser un cœur processeur soft est que nous pouvons avoir à la fois un matériel personnalisé et un processeur implémenté dans le même dispositif FPGA. Ceci implique l'utilisation du système à puce unique (FPGA), réduisant ainsi le coût du système. L'autre avantage est que le cœur soft peut être portable et peut être utilisé avec n'importe quel FPGA et n'importe quel outil de développement FPGA. Puisque le cœur soft est implémenté en logique configurable, il peut donc être réglé (configurée) en faisant varier son implémentation et sa complexité pour répondre aux besoins précis de l'application (algorithme).

Dans ce travail, nous proposons une méthode de partitionnement logiciel/matériel d'accélération de noyaux logiciels d'une application sur le dispositif logique reconfigurable (FPGA). Le processeur conçu comme un cœur soft portable utilisant VHDL et implémenté comme une partie fixe (configuré au démarrage) du FPGA, ce processeur exécute la partie non-critique de l'application en tant que logiciel. Dans la partie reconfigurable (variable) du FPGA nous implémentons le matériel qui exécute le noyau logiciel (instructions personnalisées). La partie matérielle est activée uniquement lorsque le processeur exécute l'une des instructions personnalisées. La méthode de reconfiguration partielle peut être utilisée pour basculer la partie variable à d'autres instructions personnalisées quand si nécessaire, dans ce cas, le processeur (la partie fixe) configure la partie variable à travers l'interface de configuration interne ou par l'intermédiaire d'une interface externe si l'interface interne est non disponible dans l'FPGA.

Actuellement, la partie logicielle est conçue (programmée) en utilisant le langage Assembleur. Le langage C peut être utilisé si un compilateur C est utilisé. La partie matérielle est conçue en utilisant le langage VHDL. Tout autre langage de description hardware (tels que le Verilog) peut être utilisé pour la conception de cette partie.

Bien que, l'outil logiciel ISE de Xilinx est utilisé dans le travail présenté ci-après, On essayera de garder la discussion aussi indépendante que possible de la spécificité de l'outil et de développement utilisés. Dans le code source nous utilisons seulement la construction portable du

langage VHDL; le seul module spécifique est l'instanciation de la macro bus de Xilinx lorsque la reconfiguration partielle est utilisée. L'interface de reconfiguration et les méthodes sont spécifiques pour les dispositifs cibles, elles ne sont donc pas portables sur les dispositifs et les FPGA.

Les objectifs de notre projet et la méthodologie sont de fournir une plate-forme matérielle et logicielle permettant de minimiser les efforts de conception et le temps de vérification pour toute conception future. Les efforts de conception sont minimisés par la conception du bloc matériel personnalisé (en utilisant VHDL), le temps de vérification est également réduit, car nous devons juste vérifier le bloc du matériel personnalisé.

Cette thèse est organisée comme suit : Le chapitre 1 donne les généralités et les caractéristiques des plates-formes matérielles utilisées pour exécuter un algorithme. Le chapitre 2 décrit en détail la configuration du FPGA. Dans le 3ème chapitre, nous exposons différentes métriques qui contrôlent le processus de conception d'un circuit numérique, la méthodologie et les flots de conception FPGA sont également présentés dans ce chapitre. Notre travail est présenté dans le chapitre 4, le jeu d'instructions choisi, les blocs et les phases pipelines du processeur sont présentés ici. Dans le chapitre 5, on donne les programmes de tests et les résultants des simulations des connexions de différents blocs du processeur (architectures Harvard et Von Neumann) avec et sans instructions personnalisées. Le processeur est testé en utilisant la simulation fonctionnelle et temporelle. En conclusion nous donnons des recommandations sur la façon de procéder.

Chapitre I : Les plates-formes matérielles

I.1. Introduction

Le terme "plate-forme matérielle" se réfère généralement à une configuration matérielle connue ou préalablement vérifiée qui peut être utilisée comme base pour une ou plusieurs applications. Ce chapitre est dédié aux plates-formes matérielles disponibles pour exécuter un algorithme et leurs caractéristiques. Dans la première partie, quelques généralités dans le système électronique sont présentées. Les plates-formes matérielles sont alors classées comme un processeur ayant un jeu d'instructions, ASIC basé et matériel reconfigurable. L'accent est mise sur les technologies FPGA comme matériel reconfigurable.

I.2. Généralités

I.2.1. La loi de Moore

En 1965, Gordon Moore, cofondateur d'Intel TM, fait sa fameuse observation de croissance de la densité de puces. C'est depuis dénommée «loi de Moore». Moore a observé une croissance exponentielle du nombre de transistors par circuit intégré et prédit que cette tendance se poursuivra, la figure I.1 montre l'augmentation de la densité des transistors pour les processeurs Intel [15].

Dans le monde électronique d'aujourd'hui, ces règles sont également valables pour les capacités de mémoire et la performance des systèmes informatiques. Malgré cela le coût d'un transistor à l'intérieur d'un circuit intégré continue de diminuer.

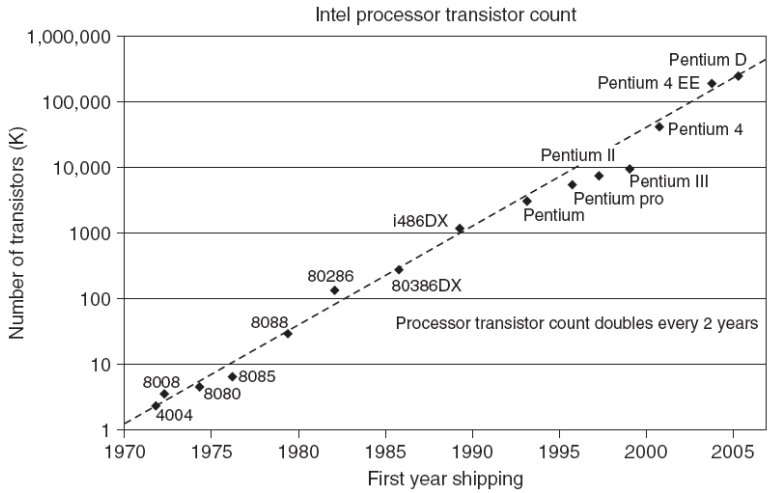


Fig.I.1 : La loi de Moore- le nombre des transistors dans les microprocesseurs Intel

Le nombre croissant de transistors dans une puce augmente le temps de vérification de la puce et augmente ainsi l'effort de conception. Pour surmonter ce problème dans les technologies modernes de nouvelles méthodes doivent être développées

I.2.2. Les métriques de conception

Le but de la conception est de construire et d'implémenter la fonctionnalité désirée, mais le défi principal de conception vient du besoin d'optimisations simultanées avec le respect de nombreuses métriques de conception. Les métriques les plus communes sont: le coût, la surface de puce, la performance, la consommation d'énergie, la flexibilité, le temps de commercialisation, et la portabilité et la réutilisabilité. La concurrence existe généralement entre les métriques de conception; l'amélioration d'une métrique peut aggraver d'autres.

I.2.2.1. La conception pour la surface

La technique la plus utilisée pour la conception pour la surface est le partage des ressources. Cette technique permet d'éviter l'allocation d'un composant matériel dédié pour chaque opération (ou fonction). Pour pouvoir partager des ressources, des multiplexeurs et des contrôleurs sont prévus pour orienter les données provenant de sources multiples vers un composant matériel unique. Les outils de synthèse doivent équilibrer la surface de circuit sauvée par le partage contre la surface introduite par ce partage, réduisant ainsi la surface matériel totale.

Le partage de ressources est moins commun dans les FPGAS en raison du coût élevé du multiplexage. Bien que le coût de partage dans les FPGAS est plus élevé que dans les ASICS. La plupart des outils de synthèse incorporent une commande de synthèse qui permet à l'outil de synthèse d'activer le partage de ressource. Le partage de ressources diminue généralement la vitesse (la performance) de la conception et particulièrement quand il est appliqué à un chemin critique (le chemin critique est le chemin qui a le plus grand retard dans un circuit). Une autre méthode pour réduire au minimum la surface est d'utiliser des modules spécifiques à la technologie. Cette méthode augmente la performance mais la conception devient moins portable.

I.2.2.2. La conception pour la performance

La performance est largement utilisée pour mesurer la qualité du système. La fréquence d'horloge est un critère commun, mais il n'est pas toujours suffisant comme mesure de performance.

Les métriques de performances les plus précises et les plus utiles sont:

- Temps de latence,
- Débit,
- Timing.

Dans le cadre de traitement des données, le débit se réfère à la quantité de données qui est traitées par unité de temps. Le temps de latence correspond au temps écoulé entre l'entrée d'une donnée et le résultat de sortie de cette même donnée. Le timing se réfère aux retards logiques (logique combinatoire) entre les bascules (les éléments séquentiels). Quand nous disons d'une conception "qu'elle ne satisfait pas le timing" cela veut dire que le retard du chemin critique est plus grand que la période d'horloge désirée [16]. Le timing détermine la vitesse maximale d'un circuit, ou la fréquence maximale, cette fréquence peut être définie selon l'équation suivante [15]:

$$F_{max} = \frac{1}{T_{clkq}+T_{maxlogic}+T_{routing}+T_{setup}+T_{skew}+T_{jitter}} \quad (I.1)$$

Où F_{max} est la fréquence d'horloge maximale; $T_{clk\text{-}Q}$ est le délai de propagation des bascules; T_{logic} est le temps de propagation de la logique combinatoire entre les bascules; $T_{routing}$ est le délai de routage entre les bascules; T_{setup} est temps d'installation (setup) des bascules, et T_{skew} est la différence maximale du temps d'arrivé d'horloge entre deux bascules; T_{jitter} est la variation de la période d'horloge.

Parceque les FPGA sont extrêmement efficaces pour l'implémentation des pipelines, exploitant des largeurs de bit non standards, et fournissant le parallélisme fonctionnel et de données [3], les deux méthodes utilisées pour augmenter la performance d'une conception FPGA sont le pipelining et le parallélisme:

1. Le Pipelining

Le Pipelining est une technique utilisé pour augmenter le débit d'une conception. L'idée fondamentale est de recouvrir le traitement de plusieurs opérations de sorte que plus d'opérations puissent être accomplies dans la même durée de temps. Si une conception peut être divisée en phases, nous pouvons insérer des bascules entre les phases et convertir une conception non pipeline en une conception pipeline. Le Pipelining offre un débit plus élevé aux dépens de la latence et les bascules interphases de pipeline (la surface).

En augmentant le nombre de phases de pipeline, la quantité de la logique combinatoire dans chaque phase se voit réduite. Toutefois, la fréquence maximale reste limitée par le temps d'installation des bascules ainsi que le temps d'arrivé et la variation d'horloge. La fréquence maximale en l'absence de logique combinatoire entre les bascules ou lorsque le nombre de phase du pipeline est très grand est donné par:

$$F_{max} = \frac{1}{T_{clkq}+T_{routing}+T_{setup}+T_{skew}+T_{jitter}} \quad \text{(I.2)}$$

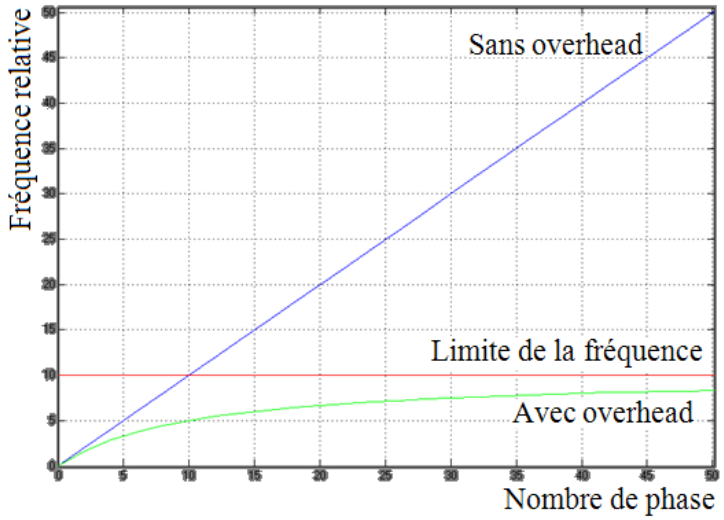


Fig.I.2 – La fréquence relative contre le nombre de phase de pipeline

La figure I.2 montre comment la fréquence maximale peut changer quand des phases de pipeline sont ajoutées. S'il n'y avait pas de temps en excès ($T_{clkq} + T_{routing} + T_{setup} + T_{skew} + T_{jitter} = 0$), passant d'une seule phase à un pipeline de 50 phases permettant une fréquence de 50 fois plus élevée, mais avec ce temps en plus, la vitesse réelle obtenue est diminué.

2. L'équilibrage registres/ logique

L'équilibrage registres/ logique (ou le retiming) est une méthode utilisée dans la conception pipeline pour équilibrer le retard entre la logique combinatoire et les bascules dans le but d'obtenir une performance maximale [3]. La plupart des outils de synthèse ont une commande de synthèse qui permet à l'outil d'effectuer le retiming. Parceque le retiming ajoute des difficultés lors de débogage, cette méthode est utilisée dans la version finale du circuit.

Un exemple de retiming est illustré à la figure I.3.

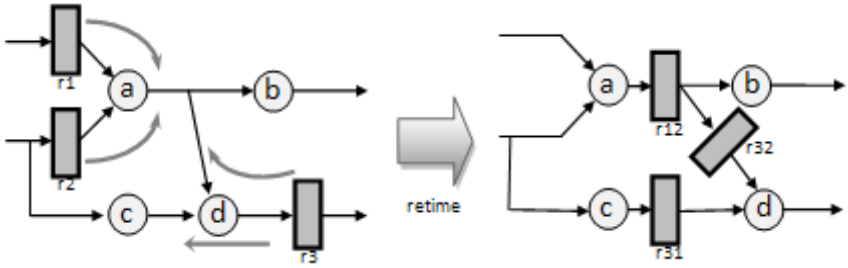


Fig I.3 - Le retiming d'un circuit - Registres r1 et r2 sont déplacés en avant pour créer des r12, et le registre -r3 est déplacé vers l'arrière pour créer r31 et r32

3. Le parallélisme

Il s'agit de la deuxième stratégie pour l'amélioration de la performance. Le Parallélisme se réfère à la répartition des données entre des différentes unités fonctionnelles. En doublant les unités fonctionnelles, on double le débit, la surface, et la consommation d'énergie. Cependant, le temps de latence reste inchangé. Le parallélisme est l'opération inverse du partage des ressources. Non tous les algorithmes peuvent profiter du parallélisme, les algorithmes qui ont beaucoup de boucles, peu de dépendance de données et dépendance de contrôle peuvent tirer profit du parallélisme, mais les algorithmes qui sont très procéduraux et contient beaucoup de branchement, ils se prêtent mal quant à l'utilisation de cette stratégie [3].

I.2.2.3. La conception pour faible consommation

Avec l'explosion du marché des appareils portables, la consommation d'énergie s'est avérée être une métrique importante. Elle permet d'éviter les emballages coûteux, et les systèmes alimentés par batterie profitant d'une autonomie accrue et d'un poids réduit. Il existe deux types de puissance dissipée dans un circuit [15] [17]: l'électricité dissipée pendant le fonctionnement

statique et pendant le fonctionnement dynamique. La puissance statique vient de courants circulant sans commutation, les courants de fuites appartiennent à cette catégorie. La puissance dynamique est le résultat des activités de commutation. Dans les circuits CMOS la puissance dynamique dissipée ***P*** est proportionnelle à la capacité de charge ***C***, la fréquence ***F***, et le carré de la tension d'alimentation V [15] [17]:

$$P = k \times C \times F \times V^2 \quad \text{(I.3)}$$

La plupart des techniques de réduction de puissance essayent d'assurer que le signal ne se modifie pas, sauf si cela est nécessaire [15] [17]. Les techniques d'économisations d'énergie vont de la simple désactivation du système en cas d'inactivité à un contrôle de puissance des composants individuels [17] [18]. Cependant, il faut noter que la puissance est très fortement liée à la performance.

Dans les conceptions FPGA toutes les techniques de réduction de puissance visent à réduire l'une des deux composantes suivantes [16]:

- La capacité ***C***: la capacité est directement liée au nombre de portes qui sont basculées à tout moment donné et la longueur des chemins de routage,
- La fréquence de fonctionnement.

Il existe une autre technique utilisée pour réduire la puissance dynamique due au basculement (glitches) est la technique d'insertion des bascules (flip-flop insertion).

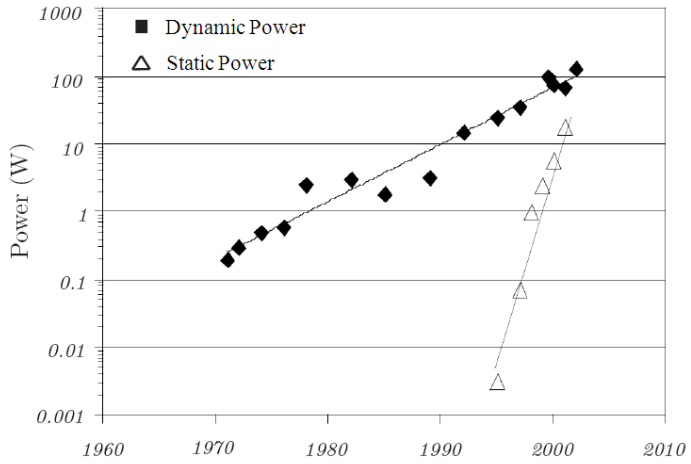


Fig I.3 – la consommation électrique dynamique et statique (fuite) d'un processeur [17]

La figure I.4 montre l'évolution de la puissance dynamique et statique (de fuite) d'une famille de processeurs [17]. La figure montre que la puissance statique deviendra l'élément dominant de la technologie moderne, et de ce fait, de nouveaux outils et méthodologie de conception doivent être développés pour réduire la puissance statique.

I.2.2.4. La conception pour faible coût

Il existe deux types de coûts dans les circuits intégrés [19]:

- Coût fixe, ce coût comprend le coût des travaux effectués par le vendeur de circuit intégré et le coût des masques plus le coût de la conception ou développement,
- Coût de fabrication de chaque copie individuelle de circuit intégré, appelé coût unitaire.

Le coût total est alors facile à calculer:

Coût total = coût fixe + coût unitaire * Quantité.

Alors, le coût final par produit est égal à:

Coût final = Total / Quantité = coût fixe / Quantité + coût d'Unité

De ce fait, une solution de compromis entre le coût fixe et celui de la quantité doit être mis en application.

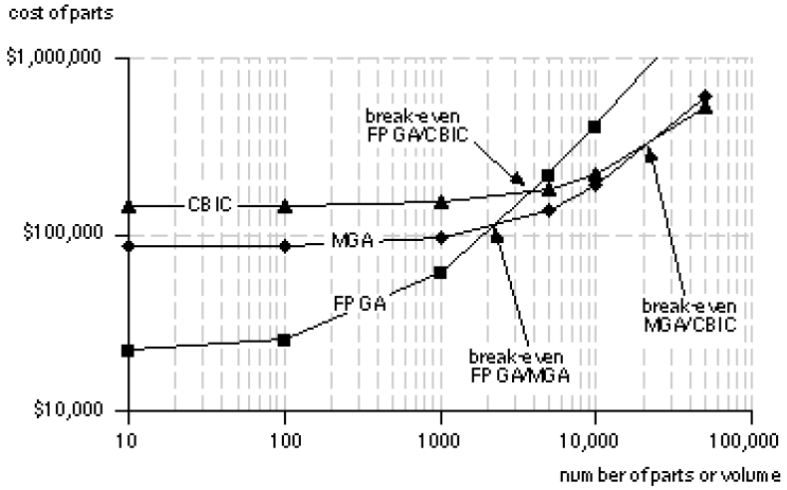


Fig I.5 - Comparaison de coût entre un FPGA, un gate array masqués(MGA) et un CBIC ASIC [19]

La Figure I.5 montre que les FPGA sont à considérer si le volume de production est petit, dans l'autre cas (volume de production important) les ASIC sont à considérer.

I.2.2.5. La conception pour le temps de mise sur le marché

Le temps de mise sur le marché est le temps nécessaire pour développer un produit jusqu'au point ou il peut être vendu à des clients. Dans l'électronique d'aujourd'hui, il est l'un des métriques les plus importantes.

I.2.2.6. La conception pour la portabilité et la réutilisation

La portabilité signifie que la même description de la conception peut être utilisée dans différentes applications pour différents dispositifs et pour presque tous les outils de développement [20]. La portabilité peut réduire le coût de développement. Elle peut également réduire la performance parceque seulement les éléments non spécifiques sont utilisés dans la description de la conception.

La réutilisation signifie que la totalité ou une partie de la description de la conception peut être utilisé à nouveau dans une autre application. Pour augmenter la réutilisation, nous devons rendre le code modulaire et extensible afin que le même code puisse être réutilisé dans différentes applications avec peu ou sans révision.

L'utilisation d'une méthodologie de conception hiérarchique permet la réutilisation. La hiérarchie est un concept important qui tend à simplifier les problèmes [17], et à réduit la taille et la complexité d'une conception schématique [18]. Elle accélère également le processus de vérification et de synthèse [21].

Pour renforcer la réutilisation d'une conception dans le cas de l'utilisation des composants spécifiques à une technologie, ces composants doivent être séparés du reste de la conception. Pour les FPGA, les mémoires sont des exemples du composant spécifique à une technologie. La plupart des vendeurs ont créé un outil pour générer automatiquement des composants spécifique. En cas de Xilinx l'outil est appelé *Core Generator* [21]; cet outil génère du modules (un fichier Electronic Data Netlist ". Edn") qui peut être déduit lors de l'étape d'implémentation. Les modules sont instanciés dans le code source VHDL. Ce qui permit de créer une boite noir lors de la synthèse en remplaçant cette dernière (la boîte noire) à l'étape d'implémentation.

Puisque les modules spécifiques doivent être remplacés lorsqu'on change de technologie et/ou d'outils, la réutilisabilité peut abaisser le coût et le temps du développement au détriment de la performance.

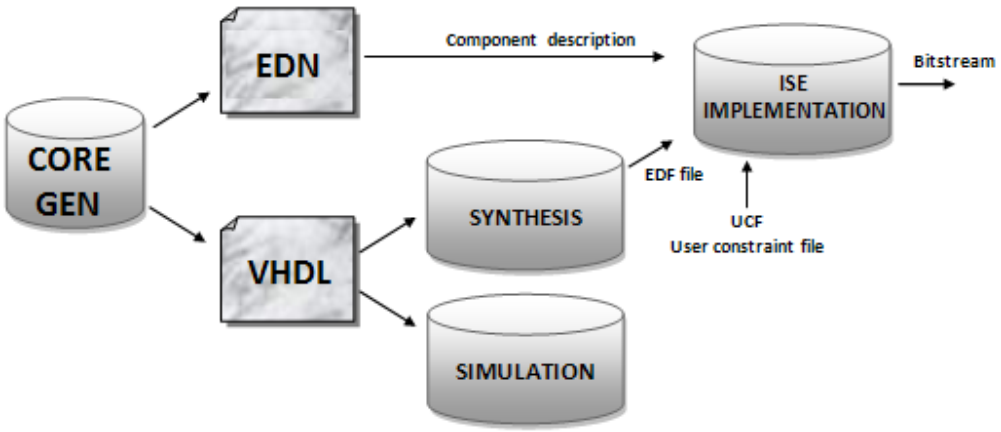


Fig I.6 - Le *Core Generator* de Xilinx permit de créer des modules FPGA spécifiques

I.3. La plate-forme FPGA

Une FPGA (Field-Programmable Gate Array) est un circuit intégré numérique qui peut être programmé (configuré) après sa fabrication. Le terme "field-programmable" se réfère à la capacité de changer le fonctionnement de composant dans le champ, tandis que "gate array" est une référence quelque peu à l'architecture interne. Un FPGA peut être vu comme un ensemble de blocs logiques programmables qui communiquent à travers une interconnexion programmable. En utilisant une configuration appropriée, les FPGA peuvent, en principe, implémente n'importe quel circuit numérique tant que leurs ressources disponibles (blocs logiques et interconnexion) sont adéquates.

I.3.1. Les ressources logiques

Il y a deux types fondamentaux de blocs logiques programmables dans les architectures FPGA [22] : architectures basées sur les multiplexeurs et architectures basées sur la table de consultation (Look Up Table ou LUT). Dans les dispositifs à base du multiplexeur le bloc logique contiennent seulement des multiplexeurs. Dans cette approche le dispositif peut être programmé tels que chaque entrée au bloc est présentée avec la logique 0, la logique 1, ou la version normale ou inverse des signaux d'entrée. Ceci permet à chaque bloc d'être configuré pour implémenter un nombre important de fonctions possibles.

Le LUT est la deuxième manière d'implémenter la logique ; le concept fondamental derrière un LUT est simple. Le N-LUT (un LUT a N entrées) peut calculer n'importe quelle fonction à N entrées en programmant simplement la table de consultation avec la table de vérité de la fonction désirée. Les différentes études ont prouvé que la structure 4-LUT fait le meilleur compromis entre la surface et la vitesse pour une large gamme d'applications [23]. Compte tenu, de l'évolution des FPGA et parceque les retards d'interconnexion est dominants dans la technologie moderne, le résultat de l'étude doit être revisité. En effet, Xilinx lance le Virtex-5 avec une architecture 6-LUT [3]. Notons que presque la totalité des FPGA commercialisé maintenant sont basés sur le LUT [3] [18].

En plus de son rôle primaire comme table de consultation, Xilinx et d'autres vendeurs autorisent les cellules de configuration formant le LUT de l'utiliser comme petit bloc de RAM appelé RAM distribuée ou comme un registre à décalage (voir la Figure I.7).

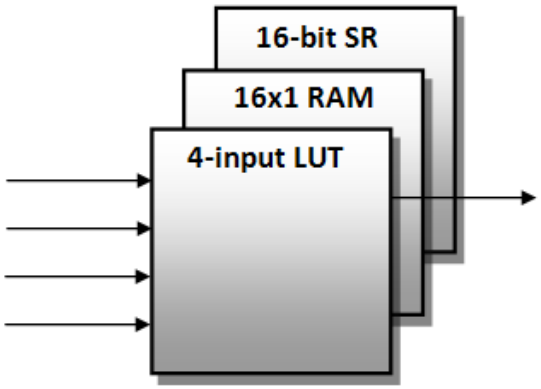


Fig I.7 – Le 4-LUT peut être configuré comme un SRAM 16X1 bits ou comme un registre à décalage de 16 bits

Les cellules logiques, Slices, et blocs logiques configurables CLB

Le bloc constitutif dans un FPGA moderne de Xilinx s'appelle cellule logique (LC). Entre autres, un LC comporte des 4-LUT, un multiplexeur, et un registre (figure I.8). Le bloc équivalent dans un FPGA d'Altera s'appelle élément logique (LE). Il y a un certain nombre de différences entre un LC et un LE, mais les concepts globaux sont très semblables. Sous Xilinx, le mot Slice se réfère à un groupement de cellules logiques. Noter bien que les LUT, les multiplexeurs, et le registre de chaque cellule logique ont leurs propres entrées et sorties de données, les cellules logiques dans chaque slice ont en commun un signal d'horloge, un signal de permission d'horloge, et un signal set/reset. Un certain nombre de slices forment un bloc logique

configurable (CLB). Certains FPGA ont deux slices dans chaque CLB, alors que d'autres ont quatre.

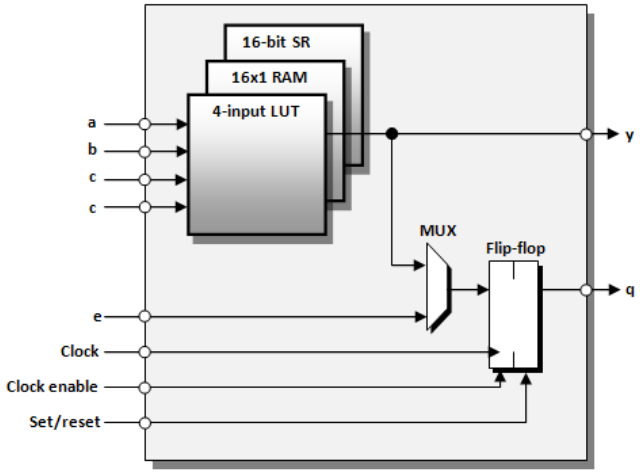


Fig I.8 – Une vue simplifiée d'une cellule logique LC de Xilinx

La chaine de propagation rapide de la retenue

Chaque cellule logique contient une logique spéciale de retenue. Elle est complétée par des interconnexions dédiées entre les cellules logiques dans chaque slice, entre les slices dans chaque CLB, et entre les CLB eux-mêmes. Cette logique spéciale et le routage dédié renforce la performance de fonctions arithmétiques telles que les additionneurs, et les compteurs.

I.3.2. Autres ressources logiques

La plupart des dispositifs FPGA incluent également certains blocs fonctionnels. Ceux-ci sont fabriqués au niveau transistor, et leurs fonctionnalités complètent les cellules logiques. Les blocs fonctionnels utilisés généralement incluent des blocs mémoire, des multiplicateurs, des circuits de gestion d'horloge, et des circuits d'interface entrée-sortie. Les dispositifs FPGA avancés peuvent même contenir un ou plusieurs processeurs préfabriqués (hard core processor).

En général, un système numérique exige la mémoire pour le stockage. Pour faciliter ce besoin, la plupart des dispositifs FPGA contiennent des modules de RAM dédiés. Tandis que ces modules ne peuvent pas remplacer les grands blocs de la mémoire externe, ils sont utiles pour les applications qui exigent une taille mémoire petite ou intermédiaire. Ces blocs ont pu être employés comme mémoire cache ou comme banque de registre dans un processeur.

I.3.3. Les ressources d'interconnexion

Le routage est la clé de la flexibilité d'FPGA, mais lui représente également un compromis fait par le vendeur de l'FPGA entre la flexibilité de programmation et l'efficacité en surface de la puce. Les interconnexions dans les FPGA sont classifiés comme programmable ou non programmable, le routage non programmable est généralement employé pour la propagation rapide de retenu pour éliminer les retards supplémentaires encouru si nous employons le routage programmable. Les interconnexions programmables sont classifiées en tant que routage local et routage global, le routage local est utilisé dans les blocs logiques pour combiner les éléments logiques dans des fonctions plus grandes et plus complexes.

Une hiérarchie d'interconnexion dans les ressources de routage globales peut être vue. Il y a des lignes longues qui peuvent être employées pour relier les CLB critique qui sont physiquement loin de l'un de l'autre sur l'FPGA sans induire beaucoup de retard. Elles peuvent également être employées comme un bus interne. Il y a également des lignes courtes qui sont employées pour relier les CLB qui sont placés physiquement près l'un de l'autre. Il y a souvent une ou plusieurs matrices de commutation, pour relier ces longues et courtes lignes ensemble. Les interconnexions programmables à l'intérieur de l'FPGA permettent la connexion des CLB aux lignes et interconnecte l'ensemble des lignes entre eux et à la matrice de commutation. Des buffers de trois-état sont employés pour relier plusieurs CLB à une ligne longue, créant de ce fait un bus. Des lignes longues spéciales, appelées les lignes globales d'horloge, sont particulièrement conçues à basse impédance et ainsi à des vitesses de propagation rapides. Celles-ci sont connectées aux buffers d'horloge et à chaque bascule dans chaque CLB. C'est comment l'horloge est distribuée dans l'FPGA.

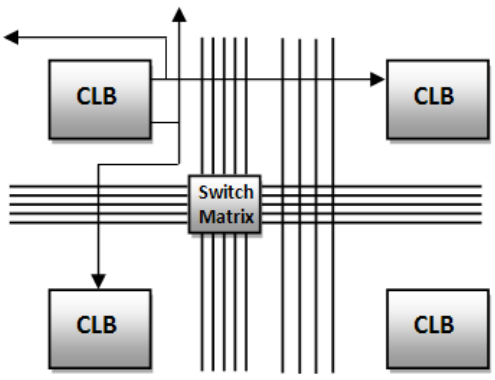


Fig I.9 – Les interconnexions de l'FPGA

I.4. La Plate-forme ASIC

Les circuits intégrés spécifiques à l'application (ou ASIC) se rapportent à ces circuits intégrés spécifiquement construits pour une application donnée. Il existe plusieurs types d'ASIC qui peuvent être classé du pleine personnalisé au semi personnalisé. Quelle que soit l'ASIC, il ait composé de modules standard, comme: portes logique, cellules de RAM, bascules, etc. Dans l'ASIC pleine personnalisé, le placement de ces modules est libre sur la puce. Les concepteurs ont la flexibilité totale, mais ceci vient au prix de développement et de coûts de production élevés. Cette technologie est seulement attrayante quand un grand nombre de même puce est produit. Une alternative meilleure marchée est les puces semi personnalisé de "sea-of-gate". En cette technologie les composants sont pré-placés. Les coûts sont réduits sensiblement parceque les portes doivent seulement être reliées par quelques couches de métal. L'inconvénient est que la conception doit être mappée sur des composants pré-placés. Par conséquent, tous les éléments ne sont pas employés parceque il est impossible de les relier tous. Ces puces seront donc plus grandes et plus lentes qu'une version pleine personnalisée, mais elles seraient meilleur marché pour concevoir et produire.

Pourquoi l'utilisation d'une solution ASIC au lieu d'une autre technologie? Il y a, en effet, beaucoup d'avantages dans l'ASIC par rapport à d'autres solutions : vitesse de fonctionnement élevé, consommation d'énergie inférieure, prix réduit pour une production en série, sécurité de conception amélioré, meilleure commande des caractéristiques d'entrées/sorties, et une conception plus compacte. Cependant, il y a des inconvénients importants : temps de mise sur le marché très long, cher pour une production de faible volume, le coût fixe est très élevé, et, finalement, une fois envoyer à la fabrication la conception ne peut pas être changée.

La conception d'ASIC continue à augmenter en taille, complexité, et coût. En même temps, la concurrence agressive rend le marché électronique d'aujourd'hui extrêmement sensibles aux pressions de temps de mise sur le marché. En outre, les fenêtres de temps du marché se rétrécissent dans le cas des marchés de consommables, N'a pas un produit disponible au début du temps prévu peut avoir comme conséquence un revenu sensiblement réduit. Ceci a conduit à la demande des méthodes de vérification rapides, efficaces, et rentables. Pour un petit circuit la simulation est une méthode efficace, mais dans le cas d'un circuit ASIC moderne, une simulation

logicielle fonctionnant sur un ordinateur à grande vitesse sera chanceuse de réaliser des vitesses de simulation équivalentes de quelques hertz, c.-à-d., quelques cycles de l'horloge pour chaque seconde en temps réel. Pratiquement, ceci signifie que les simulations logicielles détaillées peuvent être effectuées seulement sur des petites parties de la conception. Ainsi, afin de réaliser des vitesses de simulation élevée, il est nécessaire d'employer une certaine forme de vérification assistée par matériel (émulation ou prototypage FPGA).

I.5. La Plate-forme processeur

Les processeurs sont plus appropriés à l'exécution des algorithmes par rapport a d'autres plates-formes parceque ils exécutent les instructions d'un algorithme séquentiellement. Ces instructions sont stockées et cherchées de la mémoire. Puisque la modification d'un algorithme implique la modification des instructions dans la mémoire, concevoir une application sur un processeur est le plus flexible. L'exécution parallèle des instructions dans un processeur existe mais elle est limitée par la quantité de parallélisme dans l'algorithme et par le nombre d'unités fonctionnelles (unités d'exécution) disponibles dans le processeur.

I.5.1. L'architecture du processeur

L'architecture du processeur est les spécifications détaillées des opérations de calcul, de la communication, et des éléments de stockage de données dans un processeur (registres), comment ces composants agissent l'un sur l'autre (organisation de la machine), et comment ils sont commandés (jeu d'instruction). L'architecture d'une machine détermine quels calculs peuvent être exécutés (addition, multiplication…), et quels formes d'organisation de données sont employées (binaire, BCD, virgule flottante…).

Le processeur peut être divisé en chemine de données (data path) et l'unité de commande, le chemine de données assume le rôle de traitement du donnée, alors que l'unité de commande cherche les instructions, les décode, et commande les opérations du chemine de données [24].

Le jeu d'instruction se compose de tous les types d'opérations exécutables. Ils peuvent être divisés en instructions arithmétique et logique, instructions de chargement et de stockage, et instructions de commande.

Il y a deux types de jeu d'instruction CISC et RISC. Un CISC (Complex Instruction Set Computer) a des modes d'adressage multiples, des instructions de longueur variable, et de grands nombre d'instruction. Ces architectures ont l'avantage de la petite taille du programme. Cependant, seulement une petite fraction de jeu d'instruction est réellement employée dans la plupart des programmes. L'idée des architectures RISC (Reduced Instruction Set Computer) ou ordinateur de jeu d'instructions réduit est de concevoir un petit jeu d'instructions qui rendent l'efficacité de l'exécution des instructions le plus fréquemment utilisé au maximum. Les caractéristiques les plus communes des conceptions RISC sont: taille d'instruction unique, petit nombre de modes d'adressage, et aucun adressage indirect.

La différence d'implémentation du jeu d'instructions du RISC et CISC est que le processeur RISC exécute les instructions directement par le matériel où le CISC exécute les instructions à l'aide d'un microprogramme. Un microprogramme est un petit interpréteur qui prend l'instruction complexe et produit une séquence d'instructions simples qui peuvent être exécutées directement par le matériel.

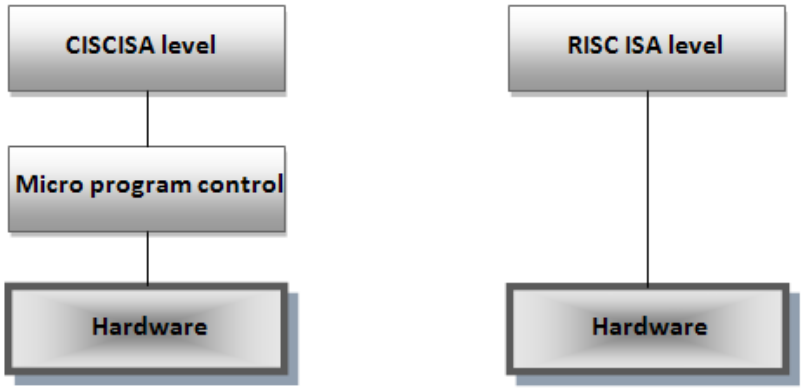


Fig I.8 – Différences entre l'implémentation CISC et l'implémentation RISC [25]

Généralement les processeurs RISC tendent d'employer une architecture spéciale connue sous le nom d'architecture de (charger/stocker). Dans cette architecture, des instructions spéciales de chargement et de stockage sont utilisées comme moyen pour déplacer les données entre les registres internes du processeur et la mémoire. Toutes les autres instructions fonctionnent seulement avec les registres internes du processeur. Les architectures RISC sont particulièrement appropriées à tirer profit de l'optimisation du compilateur, qui signifie que les programmes écrits en langages évolué sont susceptibles de voir une meilleure performance dans les environnements du processeur RISC.

I.5.2. La microarchitecture du processeur

Bien que l'architecture d'un processeur définisse les instructions qu'elle peut s'exécuter, son microarchitecture détermine la manière dont ces instructions sont exécutées. Les changements Microarchitectural ne sont pas visibles au programmeur et peuvent améliorer la performance sans amélioration du logiciel [15]. Les exemples des techniques Microarchitectural employées dans les processeurs modernes pour améliorer la performance sont l'antémémoire, le pipeline, et prédiction du branchement. L'antémémoire stocke les valeurs récemment utilisées pour réduire le temps moyen d'accès à la mémoire. Le pipeline fournit une performance plus élevé en permettant l'exécution de différentes instructions de recouvrir. La prédiction de branchement traite la pénalité d'exécution créée par les branchements dans les processeurs pipeline.

I.6. Pourquoi utilise-t-on en utilisant la plate-forme FPGA ?

Dans cette section nous donnons des raisons pour lesquelles nous choisissons la plate-forme FPGA. Les métriques de conception qui sont importante dans cette décision est le temps de mise sur le marché, la portabilité, l'effort de conception, et le coût.

I.6.1. ASIC contre FPGA

L'augmentation de temps de mise sur le marché, les coûts de développement et des masques d'AISC, de performance des FPGA et l'intégration des caractéristiques du niveau système dans les FPGA, les concepteurs d'ASIC émigrent davantage leurs conceptions à FPGA. Puisque les FPGA sont reconfigurables, ils peuvent être employés dans plusieurs conceptions sans efforts consentis [18].

La conception FPGA a un temps de mise sur le marché minimal comparée à la conception ASIC. Une fois la conception est vérifiée, la phase de production peut être lancée. En conséquence, les FPGA éliminent le temps d'attente de fabrication. Pour des productions à faible volume, les FPGA sont d'excellents candidats puisqu'ils éliminent le coût de génération de masques. Les FPGA peuvent être vérifiés dans le circuit (dans le système) sans fournir d'effort supplémentaire. La vérification matérielle peut être rapidement effectuée sur la puce. De l'autre

côté les ASIC sont employés pour une production de grand volume ou quand une grande vitesse ou une consommation très basses sont exigées.

Dans la perspective de conception, la conception FPGA n'est pas la même que la conception ASIC. Ceci fait d'elle une spécialité unique. Les techniques employées pour augmenter la performance d'un ASIC peuvent réellement diminuer la performance si utilisé dans une conception FPGA. La raison de ceci est parceque il y a une différence de performance entre les ressources d'FPGA (blocs logique, RAM, multiplicateurs, et interconnexion) et les ressources d'ASIC (portes, RAM, et fils), par exemple les interconnexions de l'FPGA sont trop lents comparés aux fils d'ASIC, mais les RAM d'FPGA presque ont la même performance que les RAM d'ASIC. La technique la plus utilisée dans la conception FPGA est le pipeline ceci est parceque les FPGA ont un grand nombre de bascules.

I.6.2. Le software (Processeur) contre l'FPGA

Tandis que les dispositifs reconfigurables (FPGA) partagent certaines des caractéristiques d'un processeur, les détails de l'architecture fondamentale sont ce qui différencie ces circuits. Les FPGA et les processeurs sont programmables mais l'organisation interne et l'utilisation de cette programmation est différente. Dans un processeur, la programmation est un ensemble d'instructions (programme) qui sont écrites dans le processeur lors du fonctionnement. Ces instructions effectuent une forme de reconfiguration. L'unité arithmétique et logique (UAL) et de divers multiplexeurs de contrôle sont configurés pour exécuter les fonctions demandées. Dans le processeur, ces composants matériels sont relativement petits et fixe et la reconfiguration est effectuée sur la base du cycle par cycle.

Les FPGA sont basés sur une structure très différente de celle d'un processeur. Il est composé d'un ensemble de blocs logiques programmables reliés entre eux par une interconnexion programmable. La plus grande différence entre un FPGA et un processeur est que l'FPGA est typiquement prévu pour être programmé comme unité complète, avec les divers composants agissant ensemble en parallèle. Les données de programmation sont utilisées dans l'espace plutôt que séquentiellement dans le temps [3].

L'avantage de vitesse des les FPGA vient du fait que le matériel programmable est personnalisable pour un algorithme particulier. Un FPGA peut être configuré pour effectuer une précision arithmétique arbitraire, avec le nombre d'unités de calcul, leur type, et leur interconnexion définie par l'algorithme. En revanche, la conception d'un jeu d'instruction fixe pour un processeur doit s'accommoder à toutes les opérations possibles qu'un algorithme pourrait exiger pour tous les types de données possibles.

Le tableau I.1 récapitule la comparaison entre les trois plates-formes.

Caractéristiques	ASIC	processeur	FPGA
Méthode de débogage	Simulation seulement	hardware	**hardware/simulation**
Temps de vérification	long	Très petit	**petit**
Effort de conception	Haut	**bas**	moyen
Performance de conception	**Haut**	bas	moyen
Temps de mise sur le marché	long	**petit**	moyen
Consommation d'énergie	**bas**	Haut	moyen
Flexibilité/portabilité	non	**oui**	**oui**
Coût de conception	Haut	**petit**	moyen

Tableau I.1-Comparaison entre les trois plates-formes (les bonnes options sont en gras)

En combinant les plates-formes de l'FPGA et du processeur nous avons obtenu une plate-forme qui a les avantages des deux plates-formes. Il y a deux choix de processeur qui peuvent être combinés avec l'FPGA, la prochaine section discute ces choix.

I.6.3. Le processeur Hard contre le processeur Soft

Il y a deux types de noyaux processeur dans l'FPGA; noyau Hard et noyau Soft. Les noyaux Hard sont intégrés dans l'FPGA au temps de fabrication, alors que les processeurs Soft sont conçus en utilisant les ressources programmables de l'FPGA. Le noyau Soft est typiquement décrit dans un langage de description matériel (HDL) ou une netlist et plus tard programmé au FPGA. Le noyau Hard offre d'excellents avantages du packaging et de communication alors que l'approche Soft offre les avantages de flexibilité et des coûts de circuits moins élevés [26] [27].

Le tableau I.2 récapitule les avantages et les inconvénients des deux noyaux.

Caractéristique	noyau Hard	noyau Soft
Performance	**Haut**	bas
Coût	Haut	**bas**
Flexibilité	-	+
Consommation d'énergie	**bas**	Haut
Nombres de processeurs dans une puce	Fixe	**Variable**
Peut être employé pour n'importe quel FPGA (portabilité)	Non	**oui**

Tableau I.2-Comparaison entre un processeur Hard et un processeur Soft (les bonnes options sont en gras)

Les noyaux soft ont peu de performance par rapport aux noyaux hard et consomment davantage d'énergie électrique. La raison de ceci est que les processeurs soft utilisent les interconnexions programmables des FPGA. Pour alléger le problème de performance est de consommation de ces noyaux soft, on peut utiliser le partitionnement hardware/software pour augmenter les performances logicielles toute en réduisant la consommation. Le partitionnement hardware/software est le processus de diviser une application entre un logiciel s'exécutant sur un processeur et un coprocesseur matériel. En identifiant les noyaux critiques contenus dans l'application logicielle, on peut ré-implémenter ces noyaux logiciel comme des coprocesseurs matériel (ou des instructions personnalisées) sur les FPGA.

I.7. Conclusion

Les métriques de conception commandent le choix de la plate-forme qui doit être employé. Le coût est une métrique importante pour n'importe quelle conception mais le temps de mise sur le marché est très important en ces jours. La conception d'ASIC est bonne pour la performance mais l'inconvénient est que le temps de développement (temps de conception et de vérification et ainsi le temps de mise sur le marché) est trop long particulièrement en technologie moderne. Pour surmonter ce problème, le dispositif reconfigurable (FPGA) peut être utilisé pour réduire le temps de mise sur le marché, car non seulement les FPGA peuvent être vérifiés par simulation, mais ils peuvent aussi être testés dans le circuit réduisant ainsi le temps de vérification, et peuvent être adaptés à une application spécifique. L'utilisation des processeurs (noyaux hard ou soft) en même temps que l'FPGA peut abaisser l'effort de conception et augmenter la flexibilité. L'utilisation des processeurs soft aide la portabilité de la conception une fois émigrée du dispositif FPGA à un autre dispositif FPGA, et à abaisser de ce fait les coûts de futures conceptions.

Chapitre II : Configuration des FPGA

II.1. Introduction

L'utilisation de la logique programmable pour accélérer le calcul, également appelé (Reconfigurable Computing) a surgi vers la fin des années 80 avec la disponibilité commerciale des FPGA. Le calcul reconfigurable est prévu pour combler la lacune entre le matériel (ASIC) et le logiciel (processeur) [23], réalisant une performance potentiellement beaucoup plus élevé que le logiciel pour beaucoup d'algorithmes [6], tout en maintenant un niveau plus élevé de flexibilité que le matériel. Ce type de calcul est basé sur les FPGA comme dispositif reconfigurable [3].

Bien que les dispositifs Reconfigurable puissent être beaucoup plus rapides que l'équivalent logiciel en particulier pour les applications qui peuvent exploiter des largeurs personnalisé de bit et le traitement parallèle spécifiques à l'application, cet avantage est diminué par le temps qu'il prend pour reconfigurée l'FPGA. La reconfiguration partielle est importante pour réduire et cacher le retard de la reconfiguration de nouvelle conception avant qu'elle puisse être exécutée.

II.2. Les Granularité des FPGA

Les FPGA sont classifiés comme à grain fin ou à grain grosse [22]. Les FPGA a grain fin ressemblent à des ASIC de rangées de porte. Dans cette architecture les blocs logiques contiennent seulement de petits éléments de base tels que les portes NON-ET, NON-OU, etc. La philosophie est que les blocs petits peuvent être reliés pour faire des fonctions plus grandes sans gaspiller trop de ressources logiques. Dans les FPGA a grain grosse, là où le bloc logique peut contenir deux bascules ou plus, une conception qui n'a pas besoin de beaucoup de bascules laissera un nombre important d'entre elles inutilisées. Malheureusement, les architectures de grain fin exigent beaucoup de ressources de routage, ce qui prend la surface et insère un grand retard d'interconnexion et consomme une grande quantité de puissance qui peut davantage que compenser l'utilisation meilleure.

FPGA Grain fin	FPGA grain grosse
Meilleure utilisation	Peu de niveaux logiques
Conversion directe en ASIC	Moins de retard d'interconnexion
Consommation élevée de puissance	Petite mémoire et petit temps de configuration [28]

Tableau II.1-Comparaison entre FPGA à grain fin contre et une FPGA a grain gros

Une comparaison des avantages de chaque type d'architecture est montrée dans le tableau II.1 ci-dessus. Le choix dont l'architecture à employer dépend de l'application.

II.3. La mémoire de configuration de l'FPGA

Chaque élément dans le bloc logique programmable dans un FPGA exige des cellules de configuration associées [3]. Le multiplexeur exige une cellule mémoire de configuration associée pour spécifier quelle entrée doit être sélectionnée. Le registre exige des cellules associées pour indiquer s'il doit agir en tant que bascule déclenchée par front ou tant que verrou sensible au niveau, s'il doit être déclenché par front montant ou front descendant d'horloge, et s'il doit être initialisé avec la logique 0 ou la logique 1. Entre-temps, le N-LUT lui-même est basé sur des cellules de la configuration de 2^N.

En outre, chaque point d'interconnexion dans la structure de routage doit être défini. C'est-à-dire, la programmation de l'interconnexion décrit comment tous les blocs logiques sont reliés. Comme dans le cas des blocs logique, les cellules de configuration sont utilisées pour configurer les chemins de câblage de l'interconnexion.

II.4. Les modèles de mémoire de configuration

Puisque la capacité de la mémoire de configuration des FPGA et le temps nécessaire pour configurer le dispositif augmentent de génération en génération, là pour une nouvelle mémoire de configuration de l'FPGA ont être développé. Il y a presque trois modèles différents de mémoire de configuration qui peuvent être employés avec les systèmes reconfigurable [29] [30] (voir la Figure II.1).

Un dispositif simple contexte exige une reconfiguration complète afin de changer une des cellules de configuration. Un dispositif multi-contexte a des couches multiples de cellules, chacun dont peut être activé à un moment donné. Un avantage de l'architecture multi-contexte par rapport à l'architecture simple-contexte est qu'il tient compte d'une commutation de contexte très rapide.

Les dispositifs qui peuvent être sélectivement programmés sans reconfiguration complète s'appellent les dispositifs partiellement reconfigurable [29]. Ces dispositifs permettent des fichiers de configurations qui occupent seulement une partie de la surface totale à configurer sur les cellules mémoire sans changer toute la configuration présente. Les petites configurations partielles ont besoin d'un temps moindre qu'une configuration complète.

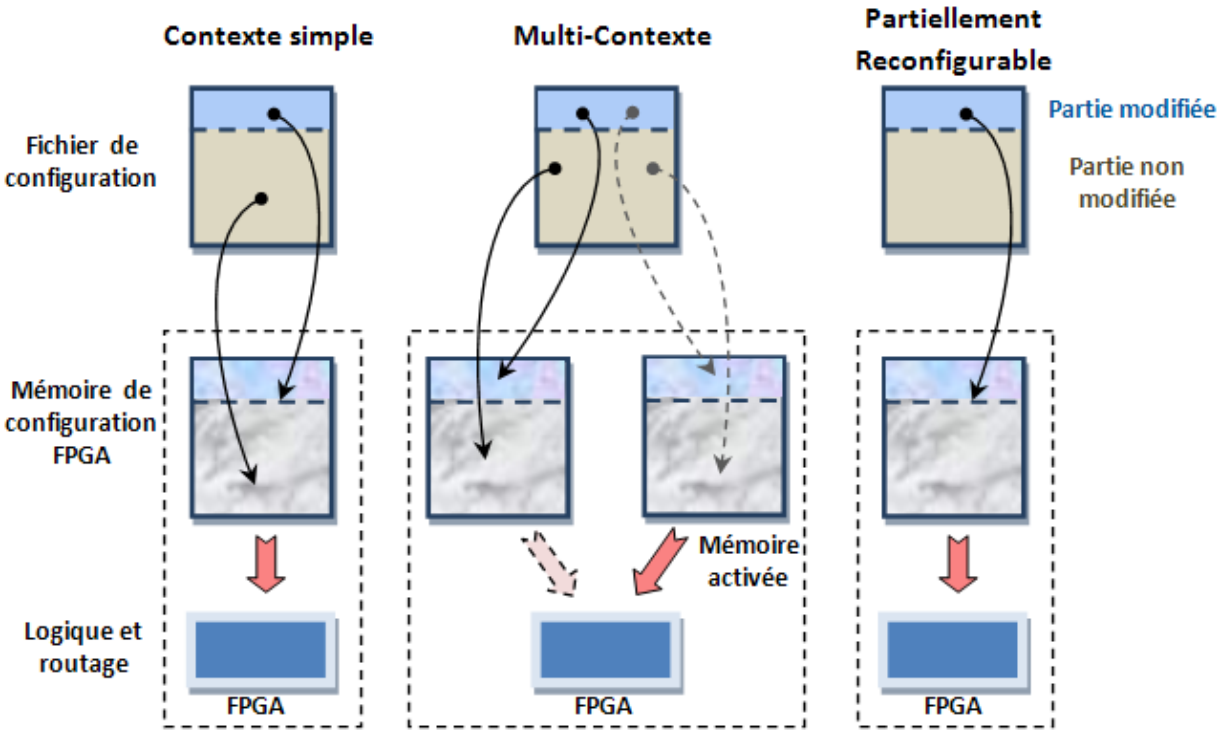


Fig. II.1 – Les différents modèles de configuration FPGA : simple contexte, multi-contexte et partiellement reconfigurable

II.5. Les Technologies de cellule de configuration des FPGA

Principalement il y a trois réalisations différentes des cellules mémoire de configuration [22]. Dans la section suivante nous décrivons ces méthodes avec leurs avantages et inconvénients.

II.5.1. Les dispositifs à base de SRAM

La majorité des FPGA sont basées sur l'utilisation des cellules de configuration type SRAM [3]. L'avantage principal de cette technologie est qu'elle est reconfigurable, ce qui fait les

dispositifs à base du SRAM les composants principal du calcul reconfigurable. Malheureusement, un inconvénient des dispositifs à base des cellules SRAM est qu'ils doivent être reprogrammés chaque fois que le système est mis sous tension. Ceci exige l'utilisation d'un circuit mémoire externe [19]. Un autre inconvénient est qu'il est difficile de protéger la conception.

II.5.2. Les dispositifs à base de l'Anti-fusible

À la différence des dispositifs à base du SRAM, ce qui sont programmables dans le système, les dispositifs anti fusible sont programmés hors système en utilisant un programmateur. Les avantages des FPGA à base d'anti fusible sont:

1) Ces dispositifs sont non-volatiles, ainsi, ils n'exigent pas le stockage externe de la configuration.
2) Bonne protection de la conception.

Naturellement, l'inconvénient principal associé aux dispositifs basés sur l'anti-fusible est qu'ils sont OTP (one time programmable), ceci laisse les composants d'anti-fusible un choix faible pour l'usage dans le développement ou le prototypage ou dans le calcul reconfigurable.

II.5.3. Les dispositifs à base de E²PROM/FLASH

Ces dispositifs peuvent être configurés individuellement hors du circuit. Alternativement, ils sont programmables dans le système, ou ISP (in system programmable). Cependant, ils ont quelques avantages:

1) Ils sont des dispositifs non-volatiles.
2) Ils sont plus sécuriser que les dispositifs à base du SRAM.
3) Les cellules E2PROM et FLASH sont plus petites que les cellules SRAM [19]. Ceci signifie que le reste de la logique peut être beaucoup plus étroit, réduisant de ce fait les retards d'interconnexion.

Du fait que ces dispositifs peuvent seulement être écrits un nombre fini de fois, ils ne sont pas le bon choix pour le calcul reconfigurable.

II.6. Les interfaces de configuration FPGA

L'interface de configuration est une interface logique entre la mémoire de configuration et le contrôleur de configuration. Généralement l'interface se compose d'un ou plusieurs pins spéciaux. Chaque interface de configuration correspond à un ou plusieurs modes de configuration [22], ces modes sont sélectionnables par l'intermédiaire des pins. Généralement il y a quatre modes de l'interface de reconfiguration et sont énuméré ci-dessous:

1. Le mode série Maître / esclave

En ce mode, l'FPGA est configuré en chargeant un bit par cycle d'horloge. Le mode de configuration série esclave tient compte que l'FPGA soit configuré par d'autres dispositifs tels que des microprocesseurs, ou par un mode de connexion en série (daisy chained). La pin d'horloge de l'FPGA est piloté par une source extérieure. En mode série maitre, l'FPGA pilote la pin d'horloge et il peut être configuré à partir d'une PROM série.

2. Le Mode Boundary-Scan

L'interface Boundary Scan (JTAG) permet l'accès en série au niveau bit à la mémoire de configuration. C'est une interface permanente qui est toujours présente dans presque tout les FPGA.

3. Le mode Parallèle Maître / esclave (Xilinx l'appelle SelectMAP [23])

Ce mode est l'option de configuration la plus rapide, il fournit un large bus de données à la logique de configuration. L'interface parallèle est typiquement commandée par un processeur, ou un autre dispositif logique tel qu'un FPGA. Des dispositifs multiples peuvent également être enchaînés en parallèle.

II.7. Les Stratégies de reconfiguration

Le matériel reconfigurable peut être configuré statiquement ou dynamiquement, là où une reconfiguration statique est seulement réalisée entre les exécutions consécutives d'une

application et une reconfiguration dynamique peut être réalisé pendant l'exécution de l'application.

II.7.1. La reconfiguration statique

La reconfiguration statique, qui désigné souvent sous le nom de la reconfiguration au moment de compilation [23], il est l'approche la plus simple et la plus utilisée. La reconfiguration statique est employée quand le taux de changement de matériel (configuration) est relativement lent : des jours ou des semaines, et le matériel configurable est assez grand pour contenir l'application. À cette stratégie, chaque application se compose d'un fichier de configuration indépendant. Cette stratégie est généralement exécutée dans le temps de départ du système. Et afin de modifier un tel système, elle doit être arrêtée tandis que la configuration est en marche et peut être remise en marche avec la nouvelle configuration matérielle. La manière économique pour implémenter cette stratégie est d'utiliser un FPGA simple contexte.

II.7.2. La reconfiguration dynamique à l'exécution

Si les parties d'un programme qui peuvent être accélérées par l'utilisation du matériel reconfigurable sont trop grandes ou complexe pour être chargées simultanément sur le matériel disponible alors il est utile de permuter différentes configurations dans et en dehors du matériel reconfigurable car elles sont nécessaires pendant l'exécution du programme, en exécutant la reconfiguration a l'exécution (runtime) [31] [23] [3].

Pour implémenter une reconfiguration à l'exécution sur des dispositifs simple ou multi contextes, les configurations doivent être regroupées dans des contextes complets, et les contextes complets sont échangés dans et en dehors du matériel selon les besoins. Les inconvénients du dispositif simple contexte sont que le dispositif doit être arrêté tandis qu'il est configuré et le temps où il prend pour être totalement configuré est trop long. Pour surmonter ce problème les dispositifs multi-contexte permettent le chargement de fond (prefetching) [23], permettant à un contexte de configurer tandis qu'un autre est dans l'exécution.

Des dispositifs partiellement reconfigurables comme les FPGA de la famille Virtex de Xilinx sont davantage convenus à la reconfiguration a l'exécution que le simple-contexte et le

multi-contexte, parceque de petites zones de la mémoire de configuration peuvent être modifiées sans exiger que la mémoire entière soit modifiée, ainsi le temps de reconfiguration est réduit au minimum. De plus, certain dispositifs FPGA permettent à une partie de la logique de configurer d'autre partie. Ce type de reconfiguration s'appelle l'auto configuration. La famille de Virtex est en particulier capable de s'auto configuré elle-même. Du fait qu'elle possède le port d'accès interne de configuration (ICAP). L'interface ICAP est semblable à l'interface maitre parallèle. Elle ne peut pas être employée pour la configuration totale du dispositif. Le bloc ICAP existe dans le bas coin droit de la puce FPGA, là pour ce coin doit appartenir à la partie non reconfigurable du FPGA. L'avantage principal de l'auto reconfiguration est que le système n'exige pas de la logique externe (extérieur FPGA) pour faire la reconfiguration partielle.

II.8. Conclusion

Les FPGA à base du SRAM sont des dispositifs qui doivent être reconfiguré chaque fois qu'ils sont mis sous tension. Ce type de reconfiguration s'appelle reconfiguration statique. Beaucoup de systèmes basés sur des FPGA initialisent leur FPGA en utilisant la reconfiguration statique mais ne changent jamais la configuration de leur FPGA au temps d'exécution quoique cette reconfiguration au temps d'exécution soit disponible dans certain FPGA et désigné sous le nom de la reconfiguration dynamique d'exécution. Les avantages de la reconfiguration dynamique sont la possibilité d'implémenter une grande application dans un petit FPGA et d'abaisser de ce fait la puissance consommée par le matériel inutilisé et le coût quand on utilise un grand FPGA (FPGA avec un nombre grand de blocs logique).

La reconfiguration dynamique est divisée en deux catégories : total et partiel. Dans la reconfiguration dynamique totale en employant le contexte simple et multiple des FPGA, tout le matériel dans l'FPGA est changé quand le dispositif est modifié. D'une part, la reconfiguration partielle est une particularité de certain FPGA qui permet la reconfiguration d'une partie de la mémoire de configuration tandis que le reste de l'FPGA continue de fonctionner. Notons que la reconfiguration partielle doit être dynamique par définition.

Chapitre III: Méthodologie et flot de conception

III.1. Introduction

Le temps nécessaire pour concevoir une petite application est très petit, tandis que l'application se développe, la longueur et le nombre de fichier source de la conception, ainsi que le temps nécessaire pour le débogage de la conception devient très important et le besoin de méthodologie de conception efficace devient évident. C'est particulièrement important quand plusieurs concepteurs sont comportés dans le même projet. Dans les sections suivantes, on propose une méthodologie de conception, cette méthodologie est basée sur la méthodologie RTL. Les objectifs principaux de la méthodologie proposée sont d'améliorer les métriques importantes suivantes:

1) La réutilisabilité et la portabilité.
2) L'effort de conception (Simplifier la vérification et la correction).
3) Minimiser le temps de conception.

La réutilisabilité peut être augmentée en explorant plusieurs propriétés importantes des HDL telles que la hiérarchie et la modularité. Pendant que la conception se développe, le temps passé sur la vérification devient une plus grande partie du temps total de la conception. Employer une méthodologie appropriée aidera certainement à réduire au minimum le temps de vérification.

Dans ce chapitre nous discuterons le niveau du détail utilise dans la conception qui est employé par n'importe quel concepteur FPGA pour implémenter une conception et pourquoi le langage VHDL est employée particulièrement au niveau RTL. Puis on donne les différents flots de conception FPGA ou les étapes que le concepteur doit suivre pour implémenter une application. Après cela, les recommandations que le concepteur doit suivre pour concevoir un processeur soft est discuté. Nous concluons ce chapitre par la méthode de partitionnement logiciel/matériel qui divise une application entre un composant logiciel fonctionnant sur un processeur et un composant matériel fonctionnant sur un matériel personnalisé.

III.2. Les méthodes et les langages de conception matérielle

Le niveau du détail dans la description de conception peut varier basé sur le niveau d'abstraction qui est employé. Un modèle plus abstrait décrira le comportement global de la

conception, mais peut exclure quelques détails internes. Un modèle moins abstrait fournira plus de détail, et soit plus près à l'implémentation réelle quand la conception est implémentée. Habituellement, les modèles plus abstraits sont plus rapides pour écrire et simuler, alors qu'un modèle plus détaillé est utilisé pour synthétiser le dispositif réel [32].

Généralement dans les conceptions FPGA il y a trois niveaux d'abstraction :

Actuellement, les plus utilisé des langages de spécifications de conception sont Verilog et VHDL au niveau de transfert de registre (RTL). Mais il y a une tendance générale vers le niveau comportemental [3] [16]. Au niveau comportemental, il n'y a aucune information temporale, ainsi le simulateur est plus rapide, et le but est habituellement d'examiner l'opération de base de la conception. Les langages utilisés dans ce niveau sont des langages de programmation logicielle comme le C, C++, SystemC, et MATLAB. En employant ces langages, on peut spécifier le comportement de la conception sans passer par une description plus détaillée, Dans ce cas-ci un outil de synthèse comportemental est utilisé pour produire des spécifications RTL. Une description plus détaillée est la description RTL. Dans de nombreux systèmes, RTL est le niveau auquel la plupart d'effort de conception et de vérification est faite. Le RTL spécifie où les registres (bascules) sont placés, fournissant les informations temporal au niveau cycle, mais pas le timing de subcycle (la logique asynchrone peut encore être abstraite), comme le temps de propagation. Ce niveau partage beaucoup de constructions avec le logiciel, mais exige plusieurs constructions matérielles pour simuler la nature parallèle du matériel. Le niveau porte est le niveau d'abstraction le plus bas utilisé dans la conception FPGA. À ce niveau, chaque élément logique individuel est spécifié. Pour une conception moyennement complexe, il est difficile de travailler avec ce niveau, puisqu'il y a tellement de détails. Les simulateurs tendent à être lents. Malgré les détails, ce niveau est abstrait, puisque les différents transistors ne sont pas décrits.

III.2.1. Le langage VHDL

VHDL comme n'importe quel langage de programmation logiciel fournit les constructions de langage de haut niveau qui permettent à des concepteurs de spécifier de grands circuits. Il contient également les caractéristiques des langages de bas niveau comme la netlist. Il supporte la création des composants pour la réutilisation dans des conceptions futures. Puisque c'est un

langage standard, et la plupart des constructions VHDL sont Synthétisable, VHDL fournit la portabilité du code entre les outils de synthèse et de simulation, aussi bien que les conceptions indépendantes du dispositif. Il facilite également la conversion d'une conception FPGA en implémentation ASIC. L'inconvénient de ce langage est que ce langage n'est pas complètement évolué, le langage s'attend à ce que toujours le concepteur connaisse quelques détails du matériel de leur conception.

Puisque la plupart des outils de synthèse ne peuvent pas synthétiser les constructions comportementales du langage, les constructions comportementales sont employées seulement dans la simulation comportementale, ainsi le langage est employé la plupart du temps dans notre travail pour spécifier la conception dans le niveau RTL.

III.2.2. La méthodologie RTL (Register Transfer Level)

Les tâches complexes sont souvent décrites par des algorithmes. Un algorithme est un ordre des étapes ou des actions. Les algorithmes sont généralement implémenter par des programmes écrits en langages de programmation séquentiels et exécutés dans un processeur d'usage général. Cependant, pour obtenir une meilleure performance et efficacité, un algorithme doit être réalisé dans un matériel personnalisé ou spécifique. La méthodologie de transfert de registre (Register Transfer Methodology) est une méthodologie de conception qui décrit le fonctionnement d'un circuit par un ordre des transferts et des manipulations de données entre les registres. Cette méthodologie peut supporter les variables et l'exécution séquentielle d'un algorithme et fournir une méthode systématique pour convertir un algorithme en matériel [20].

L'avantage de cette méthodologie n'est pas simplement limité à la synthèse. Il peut faciliter les autres tâches du processus de développement. L'impact de la méthodologie RTL est récapitulé ci-dessous [21].

1) **Synthèse**. Puisque nous pouvons séparer les éléments de la mémoire, le circuit est réduit à un circuit combinatoire.
2) **L'analyse temporale (Timing analysis)**. L'analyse implique seulement une boucle simple (à partir d'un registre à un autre registre). Ainsi, l'analyse du circuit séquentiel est essentiellement réduite à l'analyse de sa partie combinatoire.

3) **L'utilisation d'un simulateur basé sur le cycle d'horloge (simulation RTL)**. À la différence de la simulation pilotée par événements [17]. La simulation RTL ignore les retards de propagation exacte et simule l'opération de circuit d'un cycle d'horloge à un autre cycle. Nous notons que les simulateurs RTL est plus rapide que les simulateurs pilotée par événement.
4) **Réutilisation de la conception**. La contrainte temporale principale de la conception RTL est incluse dans la période d'horloge, Et ceci dépend principalement du retard de propagation de la partie combinatoire, tant que la période d'horloge est assez grande, la même conception peut être implémentée dans des dispositifs de différentes technologies.
5) **Re-implémentation de la conception dans un ASIC**. Puisque la même conception synchrone peut être visée à différentes technologies de dispositif avec un recodage minimal, d'abord il est possible de construire la conception en technologie FPGA, exécuter et vérifier le circuit à une fréquence plus basse, et puis le fabriquer en technologie ASIC.

Tous les éléments de mémoire doivent être des bascules, la description d'un verrou n'est pas permise dans cette méthodologie. L'utilisation des verrous complique l'analyse temporale et augmente ainsi le temps de vérification de la conception.

III.3. Les flots de conception FPGA

Le flot de conception est les étapes qu'on doit suivre pour produire l'implémentation de l'application ou le bitstream qui est employé pour configurer le FPGA. Une plate-forme efficace de l'FPGA, ne mènera pas à l'implémentation optimale de l'application sans un ensemble efficace d'outils pour projeter l'application sur les ressources du dispositif, vérifie sa fonctionnalité et sa contrainte du temps, et en fin, produire le bitstream pour la configuration.

Généralement il y a deux types de flot de conception FPGA typique et spécial. Le flot typique est le même pour presque tous les FPGA, ce flot et utilisé pour produire la configuration statique, le flot spécial est employé dans la reconfiguration partielle et n'est pas le même pour tout les FPGA. Les deux flots sont décrits brièvement dans les deux sous-sections ci-dessous.

III.3.1. Le flot de conception FPGA typique

Une conception typique pour FPGA implique quatre étapes, qui sont montrées sur la figure III.1, et ils sont décrits brièvement ci-dessous.

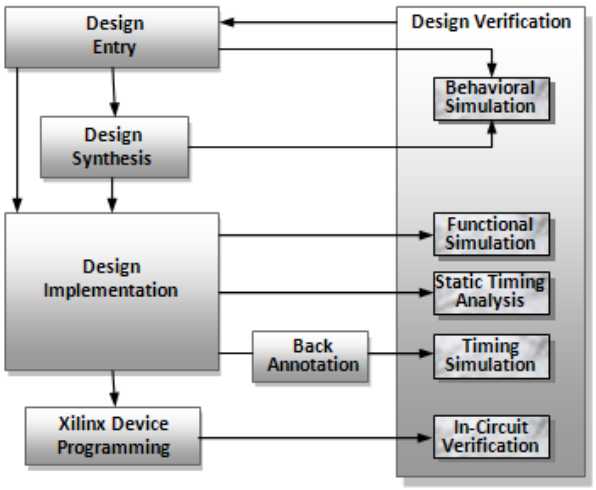


Fig.III.1- Flot de conception typique d'une FPGA [21]

1. L'entrée de la conception

Une série d'outils sont disponibles pour accomplir l'étape d'entrée de conception. Quelques concepteurs emploient l'entrée schématique tandis que d'autres préfèrent de spécifier leur conception en utilisant un langage de description matériel. D'autres préfèrent de mélanger les deux méthodes dans la même conception. Les entrées au flot de conception incluent typiquement les spécifications HDL de la conception ou le schéma, les contraintes de conception, et le dispositif FPGA cible.

- La première entrée est la description RTL de la conception. Si des spécifications plus élevées (comportementale) sont employées alors un outil de synthèse comportementale doit être utilisé pour convertir la description comportementale en spécifications RTL.
- Les contraintes de conception incluent typiquement la fréquence d'opération désirées, les retards d'acheminement entre les broches d'entrée et celles de sorties, entre les broches d'entrée et les bascules, et entre les bascules et les broches de sortie. Cependant, il faut noter que les contraintes de conception

peuvent inclure des contraintes de localisation physique, qui spécifient que certains éléments logiques soient placés à certains endroits.

- La troisième entrée de conception est le choix du dispositif FPGA cible.

2. La synthèse

Il y a deux types de synthèse matérielle, à savoir, la synthèse RTL, qui crée une netlist générique de niveau porte de la description RTL, la synthèse comportementale qui crée une description RTL à partir d'une représentation algorithmique. Dans la synthèse RTL, la description de circuit est convertie en netlist de portes logiques génériques. Ceci implique l'optimisation logique qui est une opération indépendante de technologie qui simplifie la fonction logique de la conception sans utilisation des informations spécifique à la technologie.

3. L'implémentation de la conception

Après que la conception soit écrite et synthétisée, elle est prête pour l'implémentation sur le dispositif FPGA cible. Le mappage technologique permet le mappage des portes génériques de la conception dans les composants de la bibliothèque logique du dispositif cible. Alors le logiciel groupe les primitifs dans des blocs logiques. Après, le logiciel d'implémentation cherche le meilleur placement pour placer les blocs logique parmi toutes les possibilités. Le but primaire des outils de placement est de réduire la quantité de ressources de routage exigées et de maximiser la performance de système. Quand le processus de placement et routage est fini, le logiciel crée le fichier bitstream employé pour configurer l'FPGA cible. Dans de grandes ou complexes applications, le logiciel peut ne pas pouvoir placer et router la conception avec succès. Les outils permettent au concepteur d'essayer différentes options ou d'exécuter plus d'itération afin d'essayer d'obtenir une conception plein-router. En outre, quelques fournisseurs fournissent des outils floor-planning pour aider la disposition physique des composent du circuit.

4. La vérification

Avec la croissance de complexité et la réduction du temps de mise sur le marché, le plus grand goulot d'étranglement dans la conception matériel numérique est la vérification de la

conception [17]. On signale qu'entre 40% et 70% d'effort de conception est dépensé sur la vérification [32]. La vérification se produit à de divers niveaux et étapes dans toute la conception. La simulation fonctionnelle ou comportementale est effectuée en même temps que l'entrée de conception pour vérifier l'exactitude de la fonctionnalité. La simulation temporelle doit attendre jusqu'après l'étape de placement et routage. Tandis que la simulation est toujours recommandée, habituellement l'FPGA n'exige pas la stimulation temporelle approfondie comme ASIC. Une technique réussie pour la conception FPGA est de simuler fonctionnellement la conception pour garantir la fonctionnalité appropriée, vérifiez le timing utilisant l'outil analyse statique de timing, et vérifiez alors la fonctionnalité complète en examinant la conception dans le système (dans le circuit). Les FPGA ont un avantage distinct par rapport aux ASIC. Les changements sont pratiquement gratuits. Utilisant des techniques de vérification dans le système, la conception est vérifiée à pleine vitesse, avec tous les autres composants matériel et logiciel intégrer.

III.3.2. Le flot de conception spéciale

Le flot de conception spéciale est employé dans la reconfiguration partielle. Ce flot n'est pas le même pour tout les FPGA. La section suivante donne une vue d'ensemble du flot de la reconfiguration partielle spécifique à Xillinx.

Le flot de conception de la reconfiguration partielle

La note d'application 290 [21] décrit deux méthodes pour la reconfiguration partielle. La méthode modulaire, comme représentée sur la figure III.2 divise l'FPGA en parties pour des fonctions spécifiques. Ces parties viennent avec des restrictions considérables. Ils doivent s'étendre sur toute la hauteur du dispositif. Toutes les ressources dans un module font partie de ce module et toutes les ressources inutilisées sont indisponibles pour d'autres modules. Les dimensions d'un module est fixe, ceci signifie que seulement les modules avec une taille et une forme semblables peuvent être placé dans un même espace. La communication entre les modules doit être faite par des macros bus situés aux bords du module. Le flot de conception suit le flot normal de Xilinx pour chaque module, prenant en considération quelques limitations.

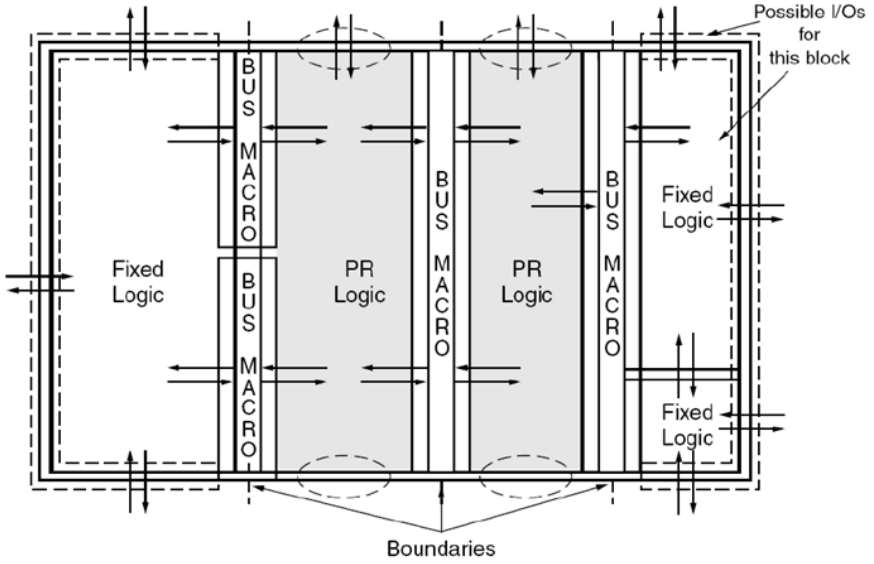


Fig.III.2- Les modules partiellement reconfigurables de Xilinx. [21]

Ces modules sont combinés dans une conception supérieure dans laquelle ils sont assignés à une surface spécifique et des informations sont fournies sur la connectivité entre les modules. Chaque module est synthétisé séparément et peut être simulé individuellement. Pour chaque combinaison des modules la conception supérieure est synthétisée et examinée, De ces conceptions un bitstream partiel est créé. Pour chaque transition entre les configurations un bitstream partiel spécifique doit être synthétisé à l'avance.

La méthode basé sur la différence, décrit comme deuxième méthode dans la note d'application, est proposée pour les petits changements de la conception. La conception est ouverte dans l'éditeur FPGA et les manipulations sont faites manuellement. Ces manipulations entourent habituellement de changer le contenu de LUT ou changeant les normes d'entrée-sortie des pins. Cette méthode n'est pas très bien adaptée pour faire de grands changements au routage. La note d'application convoque les utilisateurs à utiliser l'approche modulaire à la place. Ceci se comprend, parceque changer le routage peut mener aux dommages au dispositif en performant la reconfiguration.

III.4. Les problèmes de concevoir un processeur Soft sur FPGA

Puisque les FPGA sont limités dans l'espace (nombre limité des blocs logique), il n'est pas efficace de projeter des grandes applications entièrement au matériel. L'utilisation des

processeurs (soft ou hard) peut simplifier le processus de projection et d'économiser les ressources FPGA. Les processeurs soft occupent moins de 1% du dispositif FPGA d'aujourd'hui, ainsi les processeurs soft sont maintenant des produits courants.

Un avantage d'employer un processeur soft est que nous pouvons avoir un circuit personnalisé et un processeur implémenté sur le même FPGA et en abaissant de ce fait le coût d'employer deux circuits séparés pour le processeur et le matériel personnalisé. Une grande application inclut habituellement beaucoup de tâches différentes. Dans une plate-forme FPGA, nous pouvons employer un processeur Soft pour chaque tâche, la plupart des fournisseurs FPGA fournissent des processeurs Soft, mais ces processeurs sont seulement optimisés pour leur FPGA et ils ne sont pas ainsi portables.

Dans cette section nous discutons les aspects uniques de concevoir un processeur Soft portable d'FPGA contrairement à concevoir un processeur Hard d'ASIC. Une discussion de jeu d'instruction et les problèmes de conception, une comparaison des jeux d'instruction des processeurs Soft FPGA

III.4.1. Les occasions de la conception du processeur soft

Quoique les FPGA aient des inconvénients en termes de performance et de coût relativement aux ASIC, la flexibilité des FPGA fournit des occasions uniques dans la conception du processeur pour FPGA. Un concepteur FPGA peut changer la configuration du processeur soft si les exigences changent. Un concepteur ASIC ne peut pas procéder ainsi sans créer un nouvel ASIC.

Les processeurs ASIC tendent à avoir beaucoup de paramètres d'exécution (runtime). Les paramètres d'exécution sont changés par l'écriture dans un registre de contrôle. Les ressources d'ASIC sont fournies à l'avance pour supporter tous les comportements possibles et un registre de contrôle pour commander le comportement désiré. Un processeur Soft d'FPGA évite le coût additionnel de supporter les paramètres d'exécution en les convertissant en paramètres de génération. Les paramètres de génération éliminent les ressources supplémentaires d'FPGA exigées pour supporter les comportements multiples et le registre de contrôle associé. Les circuits

d'anti mémoire et de débogage et les instructions personnalisées sont des secteurs avec beaucoup d'occasions pour tirer profit de convertir les paramètres d'exécution en paramètres de génération.

III.4.2. Les problèmes du jeu d'instruction

La philosophie RISC conseille l'utilisation de l'architecture charger/stocker, des modes d'adressage simples, des formats d'instruction simples, et de fournir seulement les instructions utiliser par le compilateur ou ceux exigées pour soutenir un système d'exploitation. Tous les processeurs soft commerciaux suivent les principes de la philosophie RISC.

Pour créer un jeu d'instruction efficace pour les processeurs soft nous devons prendre en considération le manque d'efficacité des multiplexeurs et des ressources de routage des FPGA. Les instructions ayant la même longueur éliminent l'utilisation du bloc d'alignement à la phase de recherche d'instruction utilisée dans les processeurs CISC et suivent ainsi la philosophie RISC. Les instructions ayant la même longueur réduisent également la complexité du décodeur, mais il vivement conseillé de fournir un nombre minimal de formats d'instruction. Puisque les FPGA ont un grand nombre de bascules, elles sont bonnes pour des conceptions de type pipeline. Le processeur doit supporter le pipeline et le jeu d'instruction doit prendre en considération le pipeline du processeur.

III.4.3. Comparaison de jeu d'instruction du processeur Soft

Le tableau III.1 fournit des informations au sujet de deux exemples de jeu d'instruction du processeur soft d'FPGA des principaux fournisseurs d'FPGA. Tous les jeux d'instruction sont tout à fait semblables au jeu d'instruction MIPS [25]. Tous les processeurs ont 32 registres d'usage général de longueur de 32 bits.

	NIOS II	MicroBlaze
Nombre d'instructions	82	113
Formats des instructions	3	2
Taille des instructions	32 bits	32 bits
Instructions personnalisées	256	0
Taille des registres	32 bits	32 bits

Tableau.III.1- Jeu d'instruction du processeur Soft d'FPGA

III.5. Le partitionnement logiciel/matériel

Les FPGA sont excellentes pour implémenter les applications en tant que circuits personnalisés fortement parallèles, en portant une performance plus élevée. Cependant, les grandes applications implémentées sur un processeur peuvent être plus efficaces en taille et exigent moins d'effort de conception au dépriment d'une performance relativement faible. Dans certains cas, le mappage de l'application (dans son intégralité) en logiciel est préféré car il satisfait les exigences de performance. Dans d'autres cas, le mappage d'une application entièrement au matériel personnalisé sur FPGA peut être nécessaire pour satisfaire les exigences de performance. Dans beaucoup de cas, la meilleure implémentation se trouve quelque part entre ces deux extrémités.

Le partitionnement logiciel/ matériel, est le processus de division d'une application entre un composant logiciel exécuté dans un processeur et des composants matériels exécutés dans un matériel personnalisée pour réaliser une implémentation qui répond mieux à des exigences de performance, de taille, d'effort de concepteur, et d'autre métrique [3].

III.5.1. Les critères de partitionnement logiciel/matériel

En divisant une application entre un processeur et un matériel reconfigurable (FPGA), un certain nombre de critères doivent être évalués en plus des statistiques relatives au temps d'exécution des différentes parties de l'application pour déterminer si une tâche donnée devrait s'exécuter comme logiciel sur un processeur ou dans le matériel personnalisé. C'est ces ensembles de critères qui exigent le partitionnement logiciel/matériel.

- La localité spatiale des données est l'un des soucis pour diviser une tâche donnée. Si les données accédées ne sont pas contiguës, l'implémentation logicielle est préférée. Dans le cas contraire (cas des données mémoire contiguës) l'implémentation matérielle est à considérer.
- Le parallélisme au niveau des données est un autre critère important. Utilisant un grand nombre d'unités fonctionnelles comme composant matériel pour accélérer ces applications (applications de traitement d'images).

- La complexité de calcul limite souvent le processeur. Les algorithmes qui sont implémenté dans les FPGA comportent beaucoup de calculs, alors que les applications complexes sont souvent implémenter comme logiciel.
- La quantité de parallélisme au niveau tâche peut jouer un rôle dans le partitionnement idéal. Dans de nombreux cas, il est possible d'exécuter ces tâches en parallèle à travers des processeurs ou des composants matériels multiples.

III.5.2. Les Architectures du matériel personnalisé

Dans les systèmes qui emploient un logiciel (processeur) avec un matériel personnalisé, il y a quatre architectures différentes conçues pour l'usage dans l'implémentation du matériel personnalisé [01]. La variation primaire entre ces derniers est le degré de couplage avec le processeur principal. D'abord le matériel personnalisé peut être utilisé en tant qu'unités fonctionnelles à l'intérieur du processeur; c'est l'architecture la plus étroitement couplée et implique l'utilisation des instructions personnalisées. Deuxièmement, le matériel personnalisé peut être utilisé comme un coprocesseur. Troisièment, ils peuvent être utilisés comme un processeur indépendant dans un environnement multiprocesseur. Enfin, sous la forme la plus faiblement couplée comme une unité de traitement autonome externe. Ce modèle est similaire à celui des postes de travail en réseau, où le traitement peut se produire pour de très longues périodes de temps avec peu de communication.

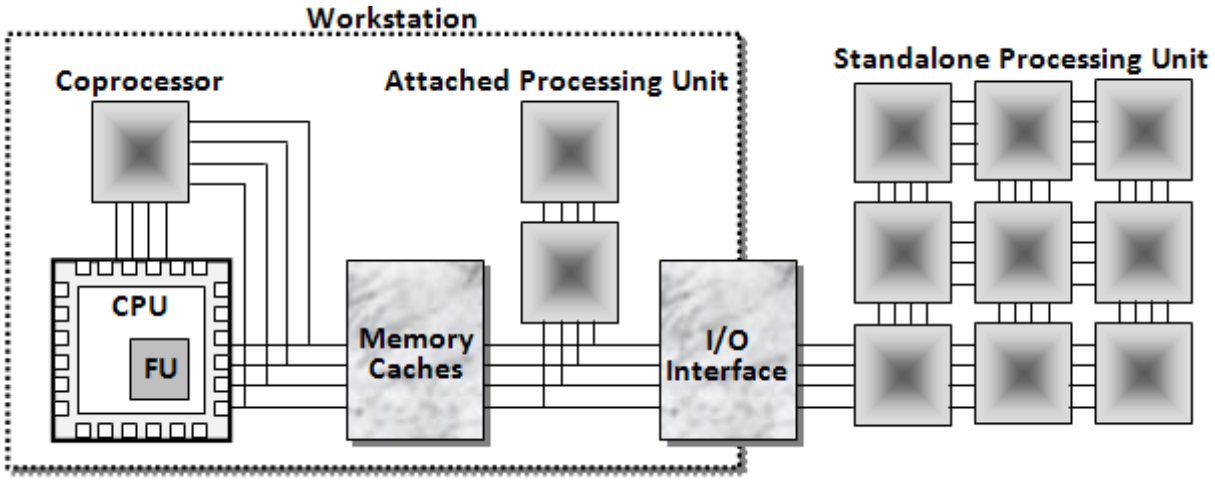


Fig.III.3- Niveaux de matériel personnalisé dans un système reconfigurable [01]

III.5.3. La personnalisation d'instructions

Les critères qui exigent l'utilisation d'une architecture ou une autre est le temps de réponse du matériel personnalisé et la simplicité de la conception de ce matériel. Par exemple l'architecture d'unité fonctionnelle est employée quand le temps de réponse de l'application est très bas et quand le matériel personnalisé doit être simple et l'effort de conception est ainsi bas. En raison de sa simplicité cette architecture est choisie dans notre travail.

```
unsigned  byteswap(unsigned x)
{
  X= (( x & 0xff00ff00) >> 8)
     |(( x & 0x00ff00ff) << 8);
  X= (( x & 0xffff0000) >> 16)
     |(( x & 0x0000ffff) << 16);
 return x;
}
```
(a)

```
Byteswap  : process ( X )  is
Begin
    Y<=   X (7 downto 0)
          & X (15 downto 8)
          & X (23 downto 16)
          & X (31 downto 24);
End process byteswap;
```
(b)

```
unsigned  byteswap(unsigned x)
{
 asm {custom x;}
 return x;
}
```
(c)

Fig.III.4- L'accélération d'une fonction C (a) par un matériel dédier (b) a accédé par une instruction personnalisé (c)

L'utilisation de l'architecture d'unité fonctionnelle signifie l'utilisation des instructions personnalisés dans la partie logicielle. La figure III.4 illustre une utilisation typique des instructions personnalisées. Le morceau de code est remplacé par une instruction personnalisé. Des recherches d'instructions et des cycles d'horloge sont économisés. De façon générale, ceci peut permettre à un plus petit processeur d'être employé, ou un matériel plus lent et meilleur marché peut devenir utilisable. Le coût, la taille et/ou la consommation du système entier peut être minimisée.

Pour personnaliser des instructions, des opérateurs matériel définis avec un langage HDL peuvent être introduit entre les entrées et les sorties du bloc personnalisé. Deux types d'opérateurs peuvent être définis. Les opérateurs combinatoires sont employés quand le retard de propagation des opérateurs est moins que la période d'horloge du processeur. Dans ce cas, les interfaces de l'unité fonctionnelle ou du bloc personnalisé sont: les bus des opérandes d'entrée et le bus ou les bus du résultat ou des résultats de sortie. Quand le retard de propagation est plus grand que la

période d'horloge, des opérations multi-cycle doivent être employées. Ils ont la même interface de données que les opérateurs combinatoires, plus l'horloge et d'autres signaux de commande.

III.6. La méthodologie

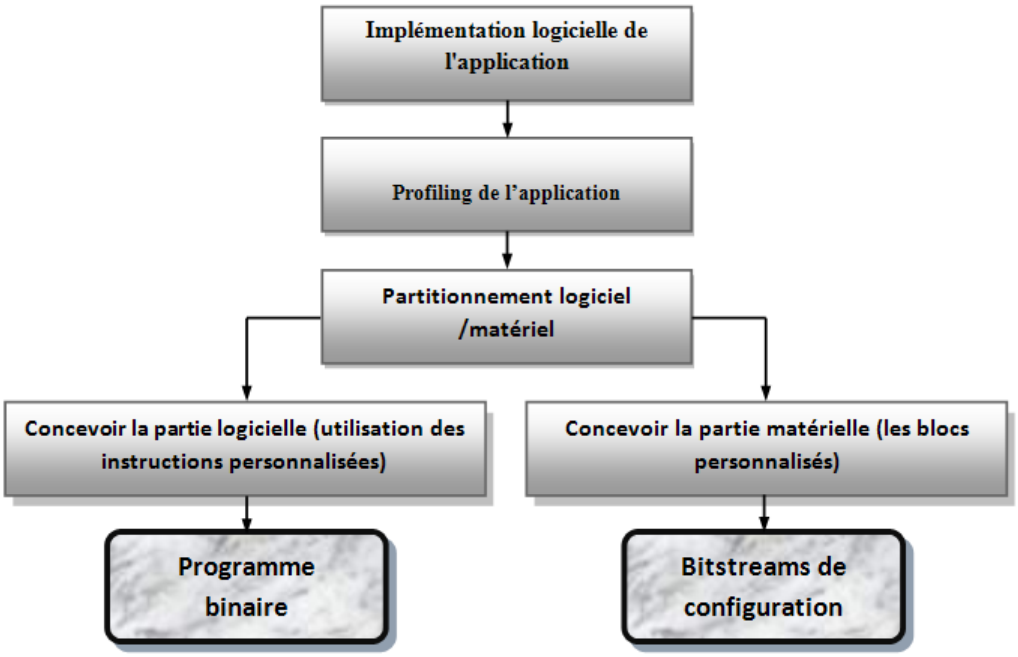


Fig.III.5- La méthodologie

Initialement l'application est implémentée comme logiciel pur. Le partitionnement logiciel/matériel est effectué en profilant cette application. Alors nous employons les informations fournies par le profiling de l'application pour extraire les noyaux candidat. Un noyau est un morceau de code qui fait passer le processeur plus de temps dans l'exécution. Pour accélérer l'application l'exécution des noyaux doit être déplacée du logiciel au matériel. Ces noyaux sont implémenté dans des blocs personnalisés qui ont des entrées et des sorties de données si l'architecture d'unité fonctionnelle est employée; l'implémentation du bloc personnalisé est effectuée en utilisant les techniques de conception matériel et le flot de conception normal de l'FPGA. Les noyaux doivent se conformer à l'interface du bloc personnalisé choisi (voir la section **Personnalisation d'instructions)**, par exemple si le bloc personnalisé choisi a deux entrées de données, les noyaux qui ont trois entrées doivent être découpés en noyaux plus simples avec seulement deux entrées. Ensuite les instructions de l'application initiale

sont modifiées en utilisant les instructions personnalisées au lieu des instructions normales pour communiquer avec les blocs personnalisés ou la partie matérielle. Le résultat de cette modification devient la partie logicielle de l'application. Les résultats de cette méthode sont les bitstreams de configuration pour les blocs personnalisé employés pour configurer le FPGA et le programme binaire que doit exécuter le processeur. Toutes ces étapes sont montrées dans la figure III.5.

III.7. Conclusion

L'utilisation des niveaux plus élevés d'abstraction (RTL ou comportemental) est exigée par la complexité de l'application. Puisque les FPGA sont limités en espace, il n'est pas efficace de mapper (projeter) de grandes applications entièrement au matériel. Les FPGA sont excellent à implémenter les applications en tant que circuits personnalisés fortement parallèles; cependant, les grandes applications implémenté comme logiciel sont plus efficace en taille et exigent moins d'efforts de conception, aux dépens d'une performance moindre. Dans les applications qui emploient des processeurs, l'utilisation du processeur Hard est plus efficace mais les processeurs Soft sont portables et flexibles. La conception du processeur Soft dans l'FPGA d'une manière ou d'autre est plus simple que la conception du processeur Hard dans l'ASIC parceque les processeurs Soft doivent être simples plutôt que efficaces. Des niveaux plus élevés de performance sur FPGA sont réalisés en employant des instructions personnalisés, accélérateurs personnalisés (coprocesseurs), ou processeurs Soft multiples. Le partitionnement d'une application entre un composant logiciel et un composant matériel est la manière pour satisfaire les exigences de la plupart des grandes et complexes applications en exécutant les parties les plus utilisé de code dans un matériel personnalisés ou reconfigurable.

Chapitre IV : Conception du processeur

IV.1. Introduction

Habituellement, la manière la plus facile de comprendre un circuit simple est d'étudier son schéma. Du schéma, nous pouvons dériver la structure équivalente de porte, les chemins critiques, ..., etc. Cependant, dès que les circuits deviennent plus compliqués, leurs schémas deviennent également plus durs pour comprendre en raison de la grande quantité des fils et des portes qui se croisent. En conséquence, il est impératif de trouver une nouvelle méthode pour décrire exactement les circuits, plus facile à comprendre et à décrire sur une quantité raisonnable de papiers. La solution pour ce problème se cache derrière le langage de description de matériel (HDL). Pour cette raison, la conception des composants du processeur est faite en suivant une approche HDL. Ensuite, nous créons un symbole schématique pour chaque composant, leur combinaison schématique forme notre processeur.

Il existe deux variétés principales de HDL: VHDL et Verilog. Les deux sont des normes standards IEEE ; cependant, Verilog est plus commun en production tandis que VHDL est utilisé la plupart du temps par les institutions de formation. Dans notre travail nous employons le VHDL comme langage de description de circuit.

IV.2. L′architecture du processeur

Dans ce chapitre, nous allons concevoir un processeur pipeline. Nous avons décidé d'utiliser le pipeline parceque les FPGA ont beaucoup de bascules, le processeur est divisé en quatre phases: recherche d'instruction (Fetch), décodage d’instructions (Dec), exécution (Exec), et Write-Back (Memwb).

1. La recherche d'instruction (Instruction Fetch):

La première phase du pipeline est l'unité de recherche d’instruction (pourrait être étendue avec le cache d'instruction), l'unité de recherche contient le registre PC, elle cherche les instructions de la mémoire en utilisant le registre PC comme adresse mémoire et à travers l'unité Ibus elle charge le code d'instructions de la mémoire et envoie l'instruction recherchée et la valeur de PC+1 à l'unité décodeur pendant le front montant d'horloge quand le décodeur est prêt.

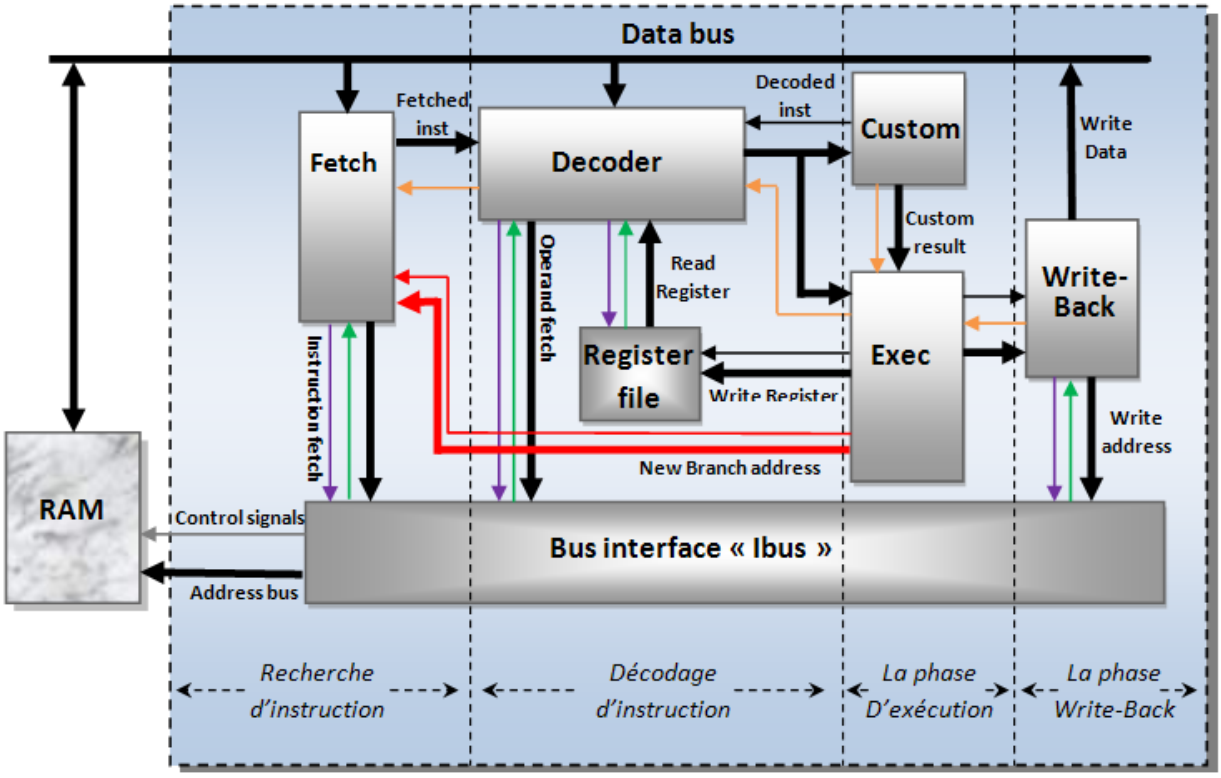


Fig.IV.1 : l'architecture pipeline du processeur

2. *Le décodage d'instruction (Instruction Decode):*

Au prochain cycle d'horloge, l'unité de décodeur cherche les opérandes de l'instruction dans la banque de registres ou dans la mémoire. Le décodeur contient un registre spécial appelé IM (the immediate register), ce registre est chargé immédiatement dans cette phase si l'instruction décodée est LOADIM, puis le décodeur place NOP (no operation) à l'unité d'exécution comme instruction décodée. Si d'autres instructions sont détectées et si les opérandes sont prêts le décodeur place l'instruction décodée et ses opérandes dans l'unité d'exécution durant le front montant de l'horloge quand l'unité d'exécution est prête.

3. *La phase d'Exécution (Instruction Execute):*

La prochaine phase se compose de l'unité d'exécution (pour instruction normale) et l'unité personnalisée (pour les instructions variable), quand l'instruction décodée est prête, l'unité d'exécution l'exécute et enregistre le résultat (le résultat vient de l'intérieur de l'unité d'exécution normale si l'instruction est normale ou à partir de l'unité personnalisée si l'instruction est

variable) dans la banque de registre ou l'envoyer avec son adresse à l'unité Write-Back. Si l'instruction décodée est un saut ou un appel ou la pin de reset est activé (sur 1 dans notre cas) ou la pin d'interruption est activée et l'interruption est acceptée l'unité d'exécution envoie alors l'adresse cible de l'instruction à l'unité recherche d'instruction comme une nouvelle adresse de recherche.

4. La phase Write-Back:

L'unité Write-Back utilise l'adresse et la donnée de l'unité d'exécution et commence un cycle d'écriture en mémoire à travers l'unité d'interface bus. La figure IV.1 montre les composants principaux et les phases du processeur. Pour la simplicité, tous les signaux dans notre processeur sont activés sur haut.

IV.2.1. Les registres du processeur

Il existe 8 registres 16 bits (R0, R1… R7) situés dans le bloc banque de registres (**regfile.vhd**). Ceux-ci sont utilisés comme registre d'usage général. Le registre R7 est utilisé comme pointeur de pile (SP). Il existe un autre registre (le registre IM) situé dans le bloc décodeur (**dec.vhd**), ce dernier est un registre 13 bits et est le seul qui peut être chargé immédiatement pendant la phase décodage. Il existe également un compteur de programme sur 16 bits (PC) situé dans le bloc de recherche d'instruction (**fetch.vhd**), ce registre est incrémenté de un (1) pour chaque instruction cherchée. Si des instructions de saut ou d'appel ou un reset ou une interruption sont exécutées le **PC** est modifié et chargé par l'adresse cible de l'opération exécutée (par exemple le 0 est l'adresse cible d'un reset). Le registre Indicateurs **Flags** est un registre 16 bits qui se situe dans le bloc unité d'exécution (**exec.vhd**), ce registre contient les drapeaux suivants: drapeau de retenue, drapeau de zéro, drapeau de signe et le drapeau d'interruption.

IV.2.2. L'architecture du jeu d'instructions

Dans notre travail, nous suivons la philosophie RISC dans la conception du jeu d'instructions. Toutes les instructions ont la même taille (pour des raisons de simplicité toutes les données et instructions sont codées sur 16 bits). Les instructions supportées sont: charger un mot

a_load, sauvegarder un mot **a_stor**, instructions arithmétiques et logiques, saut conditionnel **a_jmp**, appel, et aucune opération **a_nop**.

L'instruction **a_loadim** charge le registre **IM**, cette instruction utilise le format immédiat **LOAD_IM** ; les autres instructions de chargement ou de sauvegarde utilisent le format **MOV_MEM_REG**. Ce format utilise le contenu du registre du premier opérande **Rx** comme adresse de l'opérande mémoire. le registre destination de l'instruction de chargement ou le registre source de l'instruction de sauvegarde est soit le registre **Ry** du deuxième opérande (ex. load Rx,Ry), ou le registre **IM** (ex. load Rx,IM) ou le registre des drapeaux. Lorsque le registre R7 est utilisé comme adresse (comme premier opérande), les instructions de chargement deviennent l'instruction de dépilement (POP) sur la pile ou des instructions retour (RET), et les instructions de sauvegarde deviennent l'instruction d'empilement (PUSH) sur la pile, le registre **R7** est aussi incrémenté (dans le cas des instructions PUSH) ou décrémenté (dans le cas des instructions POP) quand il est utilisé comme adresse.

Le format **ARITH_LOG** est employé pour les instructions arithmétiques et logiques (ADD, SUB, NOT, XOR ...), le premier opérande **Rx** vient toujours de la banque de registres (voir la section *La banque de registres*) et le deuxième opérande est extrait soit de la banque de registres **Ry** ou du registre **IM**.

Le format **JMP_CALL_RET** est employé dans le cas d'un appel, d'un saut ou dans le cas des instructions de branchement conditionnel, le NOP (aucune opération) est un cas spécial du saut conditionnel où la condition est toujours fausse (aucun saut).

Le format **CUSTOM** (CUSTOM1, CUSTOM2, CUSTOM3, CUSTOM4) est employé pour les instructions personnalisées, il ressemble au format **ARITH_LOG**, le premier et le deuxième opérande sont extraits de la banque de registres **Rx**, **Ry**.

Le tableau IV.1 récapitule tous les formats d'instruction de notre processeur.

	Opcode																
Format\Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
JMP_CALL_RET-format	0	0	0	0	Condition		8bit displacement(disp8)										Conditional jump
				1	0	0	Ry_IM				RegGroup						Jump Ry/jump IM
				1	1	1	Ry_IM				RegGroup						Call Ry/call IM
LOAD_IM-format	0	0	1	13bit Immediat value (val)													Load IM with Immediat value val
MOV_MEM_REG-format	0	1	0	Rx(Adr)			Ry_IM_F_PC				RegGroup		Dir				Load register from memory Store register to memory
ARITH_LOG-format	0	1	1	Rx			Ry_IM				RegGroup		Operation				Arithmetic and logic operation
CUSTOM-format	1	0	0	Rx			Ry										Custom instruction1
	1	0	1	Rx			Ry										Custom instruction2
	1	1	0	Rx			Ry										Custom instruction3
	1	1	1	Rx			Ry										Custom instruction4

Tableau IV.1: Format des instructions supportées

Les bits 13,14, et 15 définissent le format ou l'opcode principal des instructions, il existe 8 possibilités pour l'opcode définies dans le fichier VHDL **instruction.vhd** comme des constantes, comme le montre la figure IV.2:

```
89    constant JMP_CALL_RET : std_logic_vector(2 downto 0):="000";
90    constant LOAD_IM      : std_logic_vector(2 downto 0):="001";
91    constant MOV_MEM_REG  : std_logic_vector(2 downto 0):="010";
92    constant ARITH_LOG    : std_logic_vector(2 downto 0):="011";
93
94    constant CUSTOM_1     : std_logic_vector(2 downto 0):="100";
95    constant CUSTOM_2     : std_logic_vector(2 downto 0):="101";
96    constant CUSTOM_3     : std_logic_vector(2 downto 0):="110";
97    constant CUSTOM_4     : std_logic_vector(2 downto 0):="111";
```

Fig.IV.2 : Constantes définissant l'opcode principal des instructions (instruction.vhd)

1. Le chargement immédiat LOAD_IM:

Le registre IM est situé dans le bloc de décodeur. Quand le décodeur détecte ce format dans l'instruction recherchée il charge la valeur immédiate (13 bits du champ **val**) à partir de l'opcode de l'instruction dans le registre IM et envoie un NOP à la phase d'exécution.

2. Le saut conditionnel JMP_CALL_RET:

Il s'agit du groupe d'instructions qui chargent le PC avec la valeur **PC+1+disp8** quand une condition est remplie, le déplacement **disp8** est une valeur signée de 8 bits, et par conséquent l'adresse cible de saut est entre (PC + 1-127) et (PC + 1+128). Les constantes suivantes (**instruction.vhd)** définissent les bits du champ de condition (bits 8, 9, 10, et 11). Par exemple, Si la valeur du champ condition est égale à **COND_Carry**, le saut est exécuté quand le drapeau de retenu est positionné, dans l'autre cas le saut se comporte comme une instruction NOP.

constants	valeur	signification
COND_false	"0000"	Condition toujours fausse => pas de saut=> c'est instruction de NOP
COND_true	"0001"	Condition toujours vraie (saut inconditionnel)
COND_great	"0010"	La condition est vraie quand le flag de signe est à 0
COND_less	"0011"	La condition est vraie quand le flag de signe est à 1
COND_Equel	"0100"	La condition est vraie quand le flag zéro est à 1 (le résultat est 0)
COND_NEquel	"0101"	La condition est vraie quand le flag zéro est à 0 (le résultat n'est pas 0)
COND_GrEq	"0110"	La condition est vraie quand le flag de signe est à 0 ou le flag zéro est à 1
COND_LessEq	"0111"	La condition est vraie quand le flag de signe ou le flag zéro sont à 1
COND_Carry	"1000"	La condition est vraie quand le drapeau de retenu est à 1
COND_NCarry	"1001"	La condition est vraie quand le drapeau de transport n'est pas 0
COND_OV	"1010"	La condition est vraie quand le drapeau de débordement est à 1
COND_NOV	"1011"	La condition est vraie quand le drapeau de débordement n'est pas 1

Tableau IV.2: Les conditions dans le format saut conditionnel

3. Saut et appel JMP_CALL_RET:

L'adresse cible du saut ou de l'appel est le contenu d'un des registres 16 bits (R0, R1… R7) ou le registre **IM** combiné avec les 3 bits du champ d'instruction **Ry_IM**. Le champ d'instruction **RegGroup** commande ce choix, ce champ peut avoir quatre valeurs constantes. Ces valeurs sont définies dans **instruction.vhd** comme indiqué dans le tableau IV.3.

Constant	valeur	Signification
RG_Rf	"00"	one of R0 to R7
RG_PC	"01"	the PC register
RG_IM	"10"	the IM register
RG_FLAG	"11"	flags register

Tableau IV.3 Le champ RegGroup d'instruction

Seulement les constantes **RG_Rf** et **RG_IM** sont employées dans le format **JMP_CALL_RET**, les autres constantes sont employées dans les autres formats.

4. Le format MOV_MEM_REG:

Ce format est utilisé par les instructions qui déplacent les données entre la mémoire et les registres, l'adresse de la mémoire est le contenu du registre **Rx** (R0 à R7), le champ **Dir** indique la direction du déplacement (de la mémoire au registre ou vice-versa), les champs **RegGroup** et **Ry_IM_F_PC** définissent la source (destination) d'instruction (voir le format JMP_CALL_RET).

Quand **Rx** est égal à R7 l'instruction de déplacement devient l'instruction PUSH ou POP, le champ **Dir** différencie entre les deux cas. Quand ce champ indique le sens registre vers la mémoire, l'instruction devient PUSH, sinon l'instruction devient POP dans l'autre sens de déplacement. Les instructions PUSH et POP impliquent que le registre R7 (SP) soit incrémenté (R7 = R7+1 en cas de PUSH) ou décrémenté (R7 = R7-1 en cas de POP) pendant l'exécution de ces instructions. Quand **Rx** est égal à R7 et **RegGroup** est égal à **RG_PC** et la direction du déplacement est dans le sens mémoire à registre, l'instruction devient retour (Ret ou Pop PC). Cependant, il faut noter que l'instruction RET est incluse comme un cas particulier de l'instruction POP, et les instructions POP et PUSH comme deux cas particuliers des instructions de déplacement et cela dans le but de minimiser la complexité du décodeur.

Le champ **Dir** est défini dans le fichier **instruction.vhd** comme suit:

```
144     constant MEMDIR_READ    : std_logic:='1';
145     constant MEMDIR_WRITE   : std_logic:='0';
```

et les champs **Rx,Ry** sont définis dans le même fichier **instruction.vhd** comme suit:

```
164     subtype  Ry_Type  is std_logic_vector(2 downto 0);
165     subtype  Rx_Type  is std_logic_vector(2 downto 0);
166
167     constant REGx_0       : Rx_Type :="000";
168     constant REGx_1       : Rx_Type :="001";
169     constant REGx_2       : Rx_Type :="010";
170     constant REGx_3       : Rx_Type :="011";
171     constant REGx_4       : Rx_Type :="100";
172     constant REGx_5       : Rx_Type :="101";
173     constant REGx_6       : Rx_Type :="110";
174     constant REGx_7       : Rx_Type :="111";
```

5. Le format arithmétique et logique ARITH_LOG:

Le registre indiqué par le champ **Rx** constitue le premier opérande de l'instruction, ce même registre est utilisé comme registre de destination. Le deuxième opérande de l'instruction est

indiqué par les champs **Ry_IM** et **RegGroup**, l'opérande est soit le registre **Ry** ou le registre **IM**, d'autres combinaisons ne sont pas autorisées (car elles ne sont pas implémentées).

Le champ **Operation** indique l'opération UAL, les opérations supportées sont NOT, AND, OR, XOR, SWAP échangent le contenu des octets de fort et du faible poids du registre, INC, DEC, ADC, ADD, SBC, SUB, TESTC, TEST et déplacer un registre vers un registre MOV.

6. Le format personnalisé CUSTOM:

Il existe quatre instructions disponibles avec les opcodes 100,101,110 et 111, ces opcodes sont définis dans le fichier VHDL **instruction.vhd** (c.f. figure IV.2 ci-dessus). Ces instructions ont le même format avec deux registres d'entrée **Rx**, **Ry**. Si le résultat doit être sauvegardé, **Rx** est utilisé comme registre de destination. Ces instructions sont chargées et déchargées dans l'FPGA simultanément (en d'autre termes, si nous utilisons quatre instructions personnalisées A1, A2, A3 et A4 en même temps et nous aurons besoin d'une autre instruction A5 nous devons décharger A1, A2, A3, A4 et charger A5, avec 3 autres instructions).

IV.2.3. L'Assembleur et le VHDL

Puisqu'il n'existe pas encore de compilateur de langage de haut niveau développé, nous utilisons donc un ensemble de fonctions VHDL pour remplir la mémoire d'instructions. Les entrées de ces fonctions sont les champs de l'opcode de l'instruction et le résultat est l'opcode sur 16 bits (taille du mot mémoire). Le sous-type **TDATA** défini dans **instruction.vhd** représente le mot mémoire et/ou l'opcode de l'instruction**:**

```
180    subtype TDATA IS std_logic_vector(15 downto 0);
```

Comme exemple, la fonction suivante **jmpif** renvoie l'opcode des instructions de branchement conditionnel, elle a come paramètres d'entrée la condition **cond** du type **std_logic_vector (3 downto 0)** et le déplacement **disp8** de type nombre entier. À l'intérieur de la fonction nous convertissons le déplacement en **std_logic_vector** et on utilise l'opérateur de concaténation VHDL (**&**) pour combiner les champs **JMP_CALL_RET** (l'opcode principal), le **0**(champ non utilise), **cond(3 downto 0)** et **Vdisp8 (7 downto 0)** pour construire l'opcode de l'instruction comme valeur de retour de la fonction.

Les lignes suivante est le code VHDL de la fonction **jmpif** (**instructions.vhd**).

```
318  function jmpif( cond :std_logic_vector(3 downto 0);disp8 :integer) return TDATA is
319     variable Vdisp8: TDATA;
320     variable disp_8:integer;
321    begin
322        disp_8 := disp8; if disp8<0 then disp_8 :=256 + disp8 ;end if;
323        Vdisp8 :=   std_logic_vector( TO_UNSIGNED(disp_8,Vdisp8'Length));
324        return        JMP_CALL_RET&"0"&cond(3 downto 0)&Vdisp8(7 downto 0);
325    end jmpif;
```

Pour un code plus lisible, d'autres fonctions peuvent appeler la fonction ci-dessus, par exemple **a_nop** appelle la fonction **jmpif** avec la condition **cond** ayant la valeur **COND_false** (condition toujours fausse=> aucun saut=> aucune opération) pour retourner un opcode de l'instruction sans opération(NOP).

```
339  function a_nop       return TDATA is
340        begin
341             return jmpif( COND_false ,0);
342        end ;
```

IV.3. La description VHDL du processeur

Dans les sections suivantes nous expliquons la fonctionnalité et le comportement des différents blocs du processeur. Le bloc de RAM n'est pas une partie du processeur mais il contient le programme de test et les données manipulées par ce programme.

IV.3.1. Le bloc mémoire (ram.vhd)

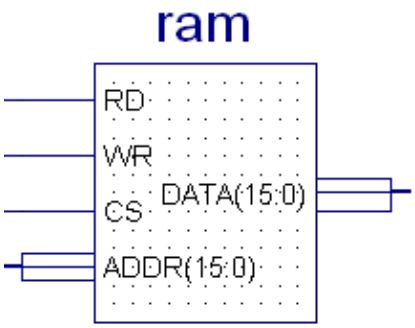


Fig.IV.3 : Symbole de RAM

Le bloc mémoire est utilisé pour tester le processeur; il contient les instructions du programme et les données manipulées par celui-ci. Les entrées du bloc sont **RD** (read), **WR** (write), **CS** (chip select), le bus d'adresses 16 bits **ADDR**(15:0), et le bus de données

DATA(15:0). La seule sortie du bloc est le bus de données, ainsi le mode de bus de données est **inout** (**in** en écrivant des données en mémoire et **out** en lisant des données à partir de la mémoire).

Les cellules mémoire internes sont représentées par un tableau nommé (signal **mem_data**) de type **TMEM_DATA**, il s'agit d'un tableau de mots de type **TDATA** (le **TDATA** est le type d'un mot mémoire). Ces types sont définis dans le fichier **instruction.vhd**.

Les définitions du mot mémoire TDATA et le tableau de mots mémoire TMEM_DATA sont:

```
180   subtype TDATA IS std_logic_vector(15 downto 0);
181   type TMEM_DATA is array (0 to memsize) of TDATA;
```

Dans les définitions, la constante **memsize** représente la taille totale de la mémoire.

Les instructions peuvent être chargées aux cellules mémoire à partir d'un fichier binaire assemblé ou compilé, l'Assembleur ou le compilateur n'étant pas encore disponibles. Pour surpasser ce problème, on fait un appel de fonction VHDL pour remplir la mémoire avec les opcodes appropriés. Dans ce qui suit, on donne un exemple de code assembleur d'un programme et son équivalent écrit dans notre VHDL.

Programme Assembleur

```
1     Load im, 10;
2     Move R1, 5;     --  reg1 =10*8+5   =85
3     Load im, 32;
4     Move SP, 0;     --  SP  = 32*8+0  =256
5     ...
```

Programme VHDL

```
1  signal mem_data: TMEM_DATA:=  -- memory cells
2     (0=>  a_loadim( 10 ),      -- address 0: we put the result of the function a_loadim( 10)
3      1=>  a_move(REGx_1, 5) ,  -- address 1: reg1 =IM*8+5  =10*8+5=85
4      2=>  a_loadim( 32 ) ,
5      3=>  a_move(REGx_7, 0) ,  -- address 3: SP   = IM*8+0 =32*8+0=256
6  ...
```

L'interaction ou le comportement de la mémoire avec l'extérieur est codée dans le processus **rdwr** (**ram.vhd**) comme suit:

```
58  IF (cs = '1') THEN                          --when selected.
59     if  (rd = '1') THEN                      --when reading.
60        data  <= mem_data(conv_integer(addr)); --the corresponding cell to outside.
61     elsif (wr = '1')then                     --when writing.
62        data <= (others=>'Z');                --we turn off the output driver
63        mem_data(conv_integer(addr)) <= data; --and store the data to the internal cell.
64     end if;
65  ELSE                                        --when not selected.
66     data <= (others=>'Z');                   --high impedance.
67  END IF;
```

IV.3.2. Le bloc d'interfaces bus (Ibus.vhd):

Il est important pour n'importe quel CPU de communiquer avec la mémoire et les périphériques. N'importe quel CPU doit utiliser une interface de bus compatible avec l'implémentation existante de bus. Nos protocoles de l'interface bus et le timing sont compatibles avec les ROM et les RAM statique et sont donc suffisants à cette fin.

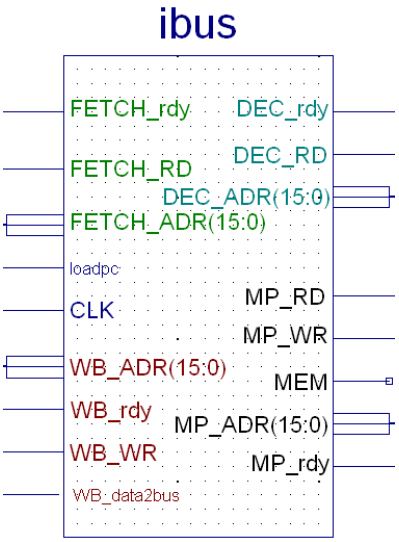


Fig.IV.4 : Symbole d'interface bus

Le but principal de ce bloc est d'éviter l'accès simultané à la mémoire par plusieurs blocs. Les accès mémoire viennent de trois blocs, un accès pour la recherche d'instructions, un autre accès pour la lecture des données (le décodeur quand il cherche un opérande mémoire), et un dernier accès pour l'écriture des données (la phase Write-Back). Un des trois blocs (fetch, dec, wbmem) est relié à l'extérieur (à la mémoire par le bus externe du processeur) à tout moment. Ce qui suit sont les signaux internes importants employés pour contrôler l'accès à la mémoire:

- Signal **cycle** : ce signal est utilisé comme moyen pour différencier entre deux états, l'état inactif (aucun accès), et l'état actif (quand il y a un accès à la mémoire).

- les signaux **iRDD**, **iRDF**, et **iWR** : ces signaux indiquent quel bloc (décodeur, bloc fetch ou le bloc Write-Back) est relié au bus externe, par exemple quand **iRDD** est actif, le décodeur est en train de chercher un opérande en mémoire. Ces signaux sont générés à partir des signaux de demande d'entrée (avec une priorité) **FETCH_RD**, **DEC_RD**, et **WB_WR** (voir le tableau IV.4).

Les signaux d'entrée			Les signaux internes		
FETCH_RD (Fetch read)	DEC_RD (Decoder read)	WB_WR (writeback write)	iRDF	iRDD	iWR
X	X	1	0	0	1
X	1	0	0	1	0
1	0	0	1	0	0
0	0	0	0	0	0

Tableau IV.4 les signaux d'entrée et les signaux internes du bloc ibus

Le code VHDL de ce bloc s'exécute comme suit:

1) Durant le front montant de l'horloge et quand le signal **cycle** est dans l'état inactif, le bloc **ibus** génère les signaux **iRDD**, **iRDF**, et **iWR** à partir des signaux d'entrées **FETCH_RD**, **WB_WR** et **DEC_RD**. Un de ces signaux est tout au plus au 1 et le bloc correspondant est relié au bus externe.
2) Durant le front descendant de l'horloge le signal **cycle** se positionne sur l'état accès si **iRDD**, **iRDF**, ou **iWR** est activé, sinon il reste dans l'état de repos.
3) Quand le signal **cycle** est dans l'état d'accès, le bloc attend l'activation du signal prêt (ready signal) **MP_rdy** à partir de l'extérieur du processeur, quand **MP_rdy** est activé, le bloc correspondant (**WB_rdy** ou **DEC_rdy** ou **FETCH_rdy**) est activé et le signal **cycle** se met au repos. Si **loadpc** est activé dans l'état d'accès (instruction de saut par exemple) on termine n'importe quel cycle de lecture (loadpc **= '1' and iWR = '0'**) parceque les données cherchées ne sont pas importantes (bloc **fetch** ou bloc **decod**), le cycle d'écriture de la donnée n'est pas terminée.
4) Quand **cycle** se met au repos, l'accès mémoire est terminé, nous allons à l'étape 1 et nous seront prêts pour un autre accès.

Une des adresses **WB_ADR** (adresse du Write-Back) ou **DEC_ADR** (adresse du décodeur) ou **FETCH_ADR** (adresse du fetch) est connectée au bus d'adresses du processeur **MP_ADR**, la connexion étant contrôlé par les signaux internes **iRDD**,**iRDF**, **iWR.**

```
87   MP_ADR <= (others=>'0') when reset='1' else
88                 WB_ADR    when iWR  ='1' else
89                 DEC_ADR   when iRDD ='1' else
90                 FETCH_ADR ;
```

Le signal **WB_data2bus** contrôle l'activation de la sortie du bloc Write-Back. Le signal **WB_data2bus** est le même que **iWR** quand il existe un accès (**cycle**='1') et '0' quand il n y a pas d'accès (**cycle**='0').

IV.3.3. Les phases pipeline du processeur

Il existe quatre phases pipeline dans notre processeur. Pour renforcer la portabilité chaque phase est codée dans un fichier VHDL séparé, ces fichiers VHDL contient l'entité qui définit l'interface avec l'extérieur de la phase et l'architecture qui définit la fonctionnalité de cette phase. Dans les prochaines sections, nous donnerons une description des différentes phases.

IV.3.3.1. La phase de recherche d'instruction (fetch.vhd)

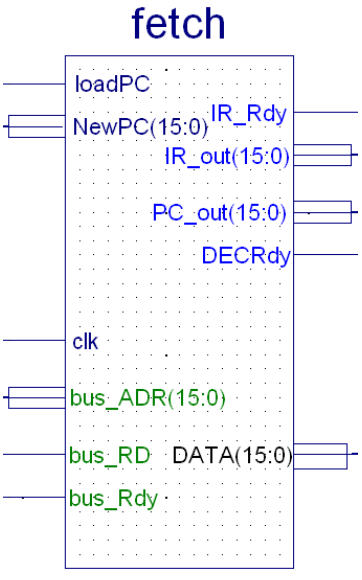


Fig.IV.5 : Symbole fetch

La phase de recherche d'instruction dirige le flot du programme et recherche les instructions de la mémoire à travers le bloc **ibus** en utilisant le registre **PC**, le signal de lecture **bus_RD**, et le

signal prêt de bus **bus_Rdy**. Cette phase sauvegarde l'instruction recherchée et son adresse correspondante (la valeur du PC +1) dans l'un des deux tampons d'instruction représentés par les signaux internes **IR_PC** et **IR_PC1**.

Les actions de remplissage ou de vidage de ces tampons sont contrôlées par une machine d'état (voir figure ci-après). Il existe trois états: l'état **SEmpty** se produit lorsqu'aucun tampon n'est rempli, l'état **SBufOne** se produit quand le premier tampon est rempli et le deuxième est vide, et l'état **SFull** apparaît quand les deux tampons sont remplis. Si nous ne sommes pas dans l'état vide **SEmpty**, cette phase conduit l'instruction et son adresse du tampon à la phase décodeur (les signaux de sortie **IR_out**, **PC_out**) quand la phase décodeur est prête (**DECRdy** ='1').

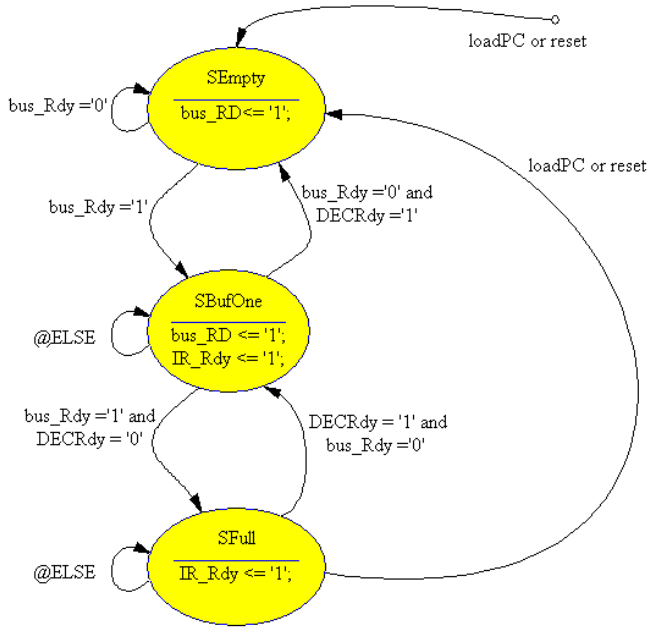


Fig.IV.6 : Le diagramme de la machine d'état du bloc fetch

L'état de la machine est contrôlé par les signaux d'entrée suivants:

1) Le signal bus prêt **bus_rdy** de **ibus**: ce signal indique qu'il y a une instruction dans le bus de données**.**
2) Le signal décodeur prêt **DECRdy**.
3) Le signal de réinitialisation globale **reset**.
4) Le signal **loadPC :** Ce signal est activé lorsqu'un saut, un appel ou un retour sont exécutées lors de la phase exécution. Si ce signal est activé, la machine d'état doit

transiter à l'état vide **SEmpty**, de ce fait, toute instruction cherchée est à ignorer et une nouvelle valeur **NewPC** est chargée dans le registre **PC**.

Le **bus_RD** (signal de lecture du bus) et **IR_Rdy** (instruction prête) sont des sorties de la machine d'état. Le signal **bus_RD** n'est pas activé à l'état **Sfull** pour empêcher de chercher plus d'instructions, et le signal **IR_Rdy** n'est pas activé à l'état **SEmpty** pour indiquer au décodeur qu'il n'y a aucune instruction présent (prêt) dans la phase de recherche.

Les tampons et le cache d'instructions

La nécessité d'avoir deux tampons est évident, quand on utilise un seul tampon et nous allons chercher une nouvelle instruction, et lorsque l'instruction est prête et le décodeur n'est pas prêt, la nouvelle instruction cherchée doit être ignorée, ce qui ralentit le processeur, De même, si l'on recherche uniquement lorsque le décodeur est prêt, ce dernier doit attendre jusqu'à ce qu'une nouvelle instruction est recherchée (ce qui ralentit le processeur). Le tampon peut être remplacé par un cache d'instruction dans de futures implémentations.

IV.3.3.2. La banque de registres (regfile.vhd):

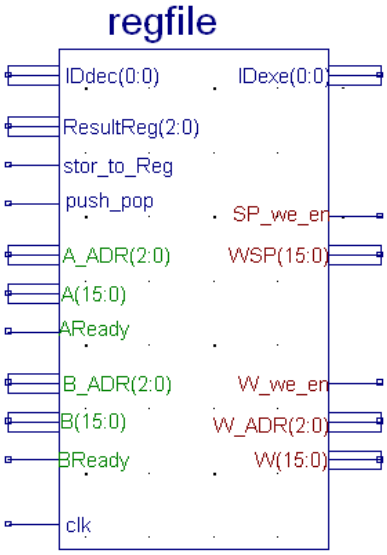


Fig.IV.7 : Symbole schématique de la banque de registres

La plupart des instructions que nous utilisons ont lisent deux registres, effectuent une opération et écrivent un résultat. Certaines d'entre elles écrivent dans deux registres (Pop et Ret

écrivent un registre normal, et elles écrivent la valeur (SP+1) ou la valeur (SP-1) dans le registre SP), la banque de registres dans notre processeur a besoin de deux ports pour la lecture et d'un port pour écrire un registre normal et un autre pour écrire dans le registre SP. Une banque de registre avec un seul port d'écriture peut être employée, mais on préfère l'utilisation de deux ports pour réduire au minimum la complexité de la phase d'exécution.

La banque de registres contient huit registres 16 bits nommés R0, R1... R7, le dernier registre R7 est utilisé comme un pointeur de pile **SP**. Tous les signaux du côté droit du bloc viennent de l'unité d'exécution et les signaux du côté gauche sont connectés à l'unité de décodeur. Quand une instruction push ou pop est exécutée, le registre **SP** est chargé via le port **WSP**(15:0) quand sa permission d'écriture **SP_WE_EN** est activé, tous les registres sont chargés via le port **W** (15:0) utilisant l'adresse de registre **W_ADR** (2:0) et la permission d'écriture de port est activé **W_we_en** quand d'autres instructions (une instruction arithmétique ou une instruction de déplacement normale) sont exécutées. Touts les chargements (l'écriture) dans la banque de registre sont synchronisés avec le front montant de l'horloge **clk**.

Le décodeur (**dec.vhd**) est connecté au côté gauche de ce bloc, nous avons deux ports de lecture A, et B. Les ports ont des sorties de données registre **A**(15:0), **B**(15:0) et 3 bits de sélection de registre **A_ADR**(2:0)**, B_ADR**(2:0) pour choisir un des huit registres (R1, R2.. R7) et les signaux de ports prêts **AReady** et **BReady**. Les signaux prêts des ports sont utilisés pour indiquer au décodeur que les registres demandés par les signaux de sélection sont prêts ou non (voir **Risque de données**).

1. L'architecture et l'entité banque de registres regfile

Le tableau ***Regs*** représente les registres R0 à R6, le registre **SP** est représenté par le signal **SP** et il n'est pas compris dans le tableau ***Regs***. L'architecture est divisée en trois partitions ou processus, le processus d'écriture, le processus de test et le processus de lecture. Dans le processus d'écriture, tout écriture ne peut être effectué que lors du front montant de l'horloge **clk**, le premier port d'écriture (utilisé pour tous les registres excepté SP) est activé quand le signal **W_we_en** est activé, le deuxième port d'écriture est utilisé seulement pour le registre **SP** et est activé par le signal **SP_we_en**, l'écriture dans le registre **SP** à partir du premier port est autorisée

quand le signal du deuxième port n'est pas activé (**SP_we_en** = '0') et le **SP** est choisi (**W_ADR** (2:0) = '111') dans le premier port. Le processus de test ***testp*** calcule les signaux prêts des ports de lecture **Aready** et **Bready**. Le processus de lecture est en réalité représenté par les deux instructions VHDL concourants dans l'architecture, ces instructions choisissent entre le tableau registre ***Regs*** quand **SP** n'est pas sélectionné et le signal **SP** quand le registre **SP** est sélectionné.

2. Risque de données et les signaux prêts

Nous avons besoin d'arrêter le pipeline du processeur afin d'éviter le problème quand une instruction écrit dans un registre le temps où la prochaine instruction lit ce même registre. Ce problème s'appelle risques de lecture après écriture [24] [25] [15]. Les risques d'écriture après écriture et écriture après lecture ne peuvent pas subvenir dans notre implémentation puisque toutes les instructions sont exécutées dans l'ordre. Notre solution est de mettre les signaux **AReady** et **BReady** à l'état non prêt quand si nécessaire; le signal **IDdec** de la phase de décodeur et le signal **IDexe** de la phase exécution sont utilisés à cette fin.

Le signal **IDdec** est un compteur d'instructions (instructions décodées). Ce compteur a un bit, cela est suffisant pour distinguer entre deux instructions consécutives. Quand l'instruction décodée est transférée à la phase d'exécution, **IDdec** est transféré à **IDexe**. Quand l'instruction finit l'exécution, deux cas peuvent se présenter:

1) Quand **IDdec** et **IDexe** sont égaux, l'instruction dans la phase de décodage et celle qui est dans la phase d'exécution sont les mêmes. Ainsi, l'instruction décodée a été exécutée et son résultat a été écrit (il n'y a aucun risque de données dans ce cas).
2) Quand **IDdec** et **IDexe** sont différents, l'instruction décodée et l'instruction exécutée ne sont pas les mêmes. Pour prévenir le risque de données dans ce cas, nous devons placer le signal prêt **AReady** ou **BReady** sur 0 (non prêt) quand le registre de destination **ResultReg**(2:0) de l'instruction qui est dans la phase d'exécution est égal au registre lu demandé de l'instruction dans la phase de décodage(**A_ADR**(2:0) ou **B_ADR**(2:0)) et le signal **stor_to_reg** est activée. Le signal **stor_to_reg** doit être utilisé parceque **ResultReg**(2:0) ne pointe pas toujours au registre de destination, par exemple l'instruction (test R1, R2) calcule la différance (R1-R2), dans ce cas

ResultReg(2:0) pointe sur le registre R1 et **stor_to_reg** n'est pas activé. Également nous devons désactiver (état non prêt) les signaux (**AReady** ou **BReady**) quand le signal **push_pop** est activé (push ,pop, call ,ret) et le registre demandé est R7 (SP) (**A_ADR**(2:0) ou **B_ADR**(2:0) égales à 111).

Le code VHDL suivant du processus **test_p** du fichier (**regfile.vhd**) récapitule ces deux cas:

```
127     AReady <= '1';                          --default values are '1' (register ready)
128     BReady <= '1';
129
130     if IDdec /= IDexe then
131
132        if ( (A_ADR = ResultReg)and stor_to_Reg ='1' ) then    AReady <= '0'; end if;
133        if ( (ASel  =7)         and push_pop    ='1' ) then    AReady <= '0'; end if;
134
135        if ( (B_ADR = ResultReg)and stor_to_Reg ='1' ) then    BReady <= '0'; end if;
136        if ( (BSel  =7)         and push_pop    ='1' ) then    BReady <= '0'; end if;
137
138     end if;
```

IV.3.3.3. La phase Write-Back (memwb.vhd):

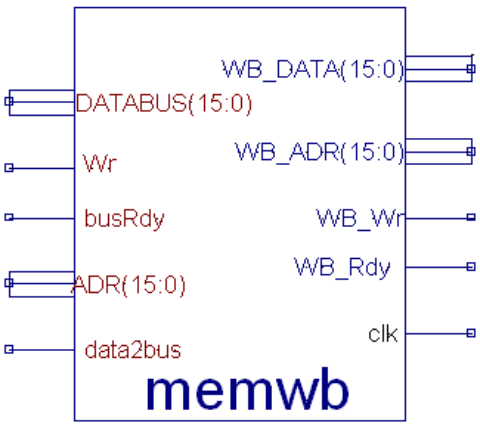


Fig.IV.8 : Symbole schématique de Write-Back

Cette phase est contrôlée par une machine d'état, il y a deux états, l'état prêt **SReady** et l'état écrire **SWriting**. L'état prêt se produit lorsque la phase ne fait rien, et l'état d'écriture se produit lorsque la phase écrit quelque chose à la mémoire. Le type VHDL **TSTATE** définit la machine d'état, et les signaux **State** et **NextState** représentent l'état actuel et l'état futur de cette machine d'état.

```
52     type          TSTATE is (SReady,SWriting);
53     signal        state,NextState :TSTATE:=SReady;
```

Le processus suivant **NextstateLogic** est la partie combinatoire de la machine d'état (figure IV.9), ce processus place le signal d'état future **NextState** aux états (**SWriting**, **SReady**) ou rester dans l'état actuel (**NextState** <= **state** ;) selon les signaux d'entrée (**WB_Wr**: demande d'écriture de la phase d'exécution, **busRdy**: bus prêt de l'interface bus) et de l'état actuel (**state**). Comme exemple quand l'état actuel est égal à **SWriting** et **busRdy** est à 1, l'état futur est placé à **SReady**.

```
60  NextstateLogic:process (state,WB_Wr,busRdy)
61    begin
62
63       NextState  <= state;  -- staying in the current state when nothing happened
64
65        CASE state IS
66
67          WHEN SReady   =>  if( WB_Wr  ='1' )then  NextState  <= SWriting;  end if;
68          WHEN SWriting =>  if( busRdy ='1' )then  NextState  <= SReady;    end if;
69
70        END CASE;
71
72  end process;
```

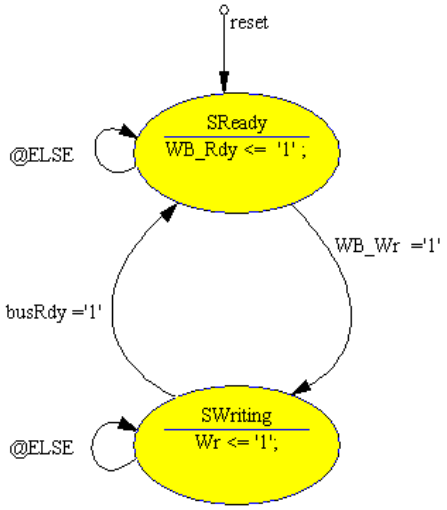


Fig.IV.9 : La machine d'état du Write-Back

Dans le processus **Statelogic** nous transférons le signal d'état futur au signal d'état actuel (state < = NextState ; l'état futur devient l'état actuel) au front montant de l'horloge (clk'event et clk = '1 ').

Les signaux **WB_Rdy** et **Wr** sont des sorties de cette machine d'état, comme exemple le signal **Wr** est placé à 1 quand la machine d'état est dans l'état d'écriture **SWriting**.

```
85  WB_Rdy   <=   '1'   WHEN State = SReady   ELSE  '0';
86  Wr       <=   '1'   WHEN State = SWriting ELSE  '0';
```

Le bus de données est piloté par ce bloc quand le signal d'entrée **data2bus** (du bloc ibus) est activé, les données de sortie sont égale aux données écrites par l'unité d'exécution **WB_DATA**. Quand le signal **data2bus** n'est pas activé le bus de données est piloté à l'état haute impédance (others=>'Z ‘).

```
88   DATABUS  <= WB DATA WHEN data2bus='1'     ELSE (others=>'Z');
```

L'adresse **ADR** allant au bloc ibus est toujours égale à l'adresse de sortie de l’unité d’exécution **WB_ADR**.

```
90   ADR      <= WB_ADR;
```

IV.3.3.4. La phase de décodage (dec.vhd):

La phase de décodage d'instruction tient la plupart de la logique de contrôle du pipeline, car elle décode ce qu’il faut pour chaque instruction. La phase de décodeur décode l'opcode d'instruction 16 bits (**IR**(15 downto 0)) venant de la phase de recherche d’instruction et met l'instruction décodée à la prochaine phase (phase d'exécution). Si l'instruction a besoin d'un opérande mémoire, cette phase cherche cet opérande en utilisant les signaux **RD**, **ADR**, **DATABUS**, et **busRdy**, si les opérandes de l'instruction proviennent de la banque de registres (R1, R2...R7), le décodeur lit ces opérandes en utilisant les signaux **ARfile_ADR**, **BRfile_ADR** (première et deuxième adresse registre), **ARfile_DAT**, **BRfile_DAT** (première et deuxième données registre), **ARfile_Rdy**, **BRfile_Rdy** (premier et deuxième signal prêt registre venir du bloc banque de registres). Si le deuxième opérande est le registre IM cette phase met la valeur du **IM**(13 bits) enchaîné avec les 3 bits du champ **Ry_IM** de l'opcode d'instruction comme opérande. Si l'instruction a besoin de la prochaine adresse PC (instructions de saut) le décodeur met l'adresse de sortie venant de la phase de recherche d’instruction à la phase d'exécution comme opérande.

Le tableau IV.5 donne une brève description des signaux décodés, dans ce tableau le signal **ExecVector** est une collection de différents signaux ; la signification et la position des différents signaux à l'intérieur de ce signal sont définies dans **instruction.vhd** et ils sont donner au tableau IV.6. Comme exemple le constant EXE_loadFlag est un nombre entier de valeur 0 ; et ceci

signifie que le signal de position correspondant ExecVector[0] est un signal de commande qui charge le registre des drapeaux.

Signal	Largeur	Signification
Reg_Rx	16	données de premier opérande (toujours à partir de la banque de registres)
ADR_Aoperand	16	données du premier opérande
Boperand	16	données du deuxième opérande
ALU_Op	4	code de l'opération d'UAL
ResultReg	3	L'adresse de registre de destination (résultat)
condition	4	La condition des instructions de branchement conditionnel
Stor_RResult	1	stocker le résultat à la banque de registres
PushPopR	1	empilement ou dépilement des données à ou de la pile
Mem_DIR	1	direction du mouvement des données mémoire (‘1’: lecture,’0’: ecriture).Ce signal est employé avec le signal PushPopR pour différencier entre le push et le pop
IDdec	1	Représenter l'identification d'instruction, ce signal bascule chaque fois qu’une instruction est décodé, ce signal est utilisé en même temps que le signal IDExec (sortie de la phase d'exécution) pour différencier entre deux instructions consécutives.
ExecVector	6	Représenter un ensemble de 6 signaux (voir le tableau ci-dessous).
Dec2ExeRdy	1	L'instruction décodée est prête.
Exec2DecRdy	1	Le signal prêt de l'unité d'exécution (phase exec).
loadpc	1	De la phase d'exécution, ce signal est activé quand la phase d'exécution exécute un saut ou un appel.

Tableau IV.5 Les signaux de sortie du décodeur

Constant	Valeur	Signal	Signification
EXE_loadFlag	0	ExecVector[0]	Charger le registre des drapeaux de la mémoire
EXE_saveFlag	1	ExecVector[1]	stocker le registre des drapeaux à la mémoire
EXE_Load_PC	2	ExecVector[2]	Charger le registre PC (saut, appel,…)
EXE_PushLoad_PC	3	ExecVector[3]	empilement de registre PC (instruction d'appel)
EXE_Cond_Jump	4	ExecVector[4]	Sauter si la condition est vraie
EXE_custom	5	ExecVector[5]	Exécuter une instruction personnalisé (et stocker probablement le résultat du bloc custom)

Tableau IV.6 les signaux à l'intérieur du signal ExecVector

Le signal **CustomSelect** est activé quand une des instructions personnalisé est décodée, le signal **CustomInst** indique lequel est choisi (0, 1, 2, ou 3). Le signal **CUSTOM_ReadRegA** est un signal de 4 bits venant du bloc personnalisé, chaque bit de ce signal est employé pour chaque instruction personnalisé pour indiquer que cette instruction a besoin du premier opérande Rx ou non, de même le signal **CUSTOM_ReadRegB** est employé pour le deuxième opérande, et le troisième signal **CUSTOM_WriteReg** de 4 bits indique pour chaque instruction personnalisé que le registre A (le premier opérande) est utilisé comme destination ou non.

Ce qui suit est une description des blocs et du différents processus de l'architecture du décodeur :

1. Le champ d'op code de l'instruction et le mot-clé VHDL ALIAS:

Le mot-clé ALIAS est employé dans la partie déclarative de l'architecture pour raccourcir différents champs de signal **IR**. Par exemple **IR_condition** est un champ de 4 bits extrait à partir de l'instruction de la position 8 à 11.Tous les noms préfixés par IR_ sont ALIAS de signal **IR**.

2. L'opérande mémoire (le signal interne Need_Mem_Op) :

La phase de décodeur doit chercher un opérande mémoire quand des instructions de mouvement sont décodées (le champ principal d'op code **IR_opcode** égal à **MOV_MEM_REG**) et la direction du mouvement indique la direction de lecture (le champ **IR_moveDir** est égal à la constante **MEMDIR_READ**). L'adresse de l'opérande mémoire est toujours la sortie de données du premier port du banque de registres (ADR < = ARfile_DAT ;).

3. Le déplacement du saut conditionnel (signal IR_disp8):

Ce signal est employé seulement dans les instructions de branchement conditionnel. Puisque nous utilisons l'arithmétique 16 bits, ce signal de 8 bits (champ disp8 de l'instruction) doit être étendu à un signal 16 bits.

```
134   IR_disp8      <= sxt( IR(7 downto 0) ,16);
```

4. Le processus processdecode:

Ce processus prend l'opcode (signal **IR**) et les entrées de cette phase (**ARfile_DAT**, **BRfile_DAT**, **ARfile_Rdy**, **BRfile_Rdy**, **busRdy**, **nextPC**, **CUSTOM_WriteReg**, **CUSTOM_ReadRegA**, **CUSTOM_ReadRegB**, **decinterrupt**) et les signaux générés en interne (**Need_Mem_Op**, **DATA,im15to3**) pour produire le signal de sortie combinatoire **IRdec** (**IRdec** est une collection (record de VHDL) de différents signaux définis dans **instruction.vhd**). Ces signaux (**IRdec**) sont transférés à la phase d'exécution dans le processus **clockedout** quand la phase d'exécution est prête.

5. La machine d'état de décodeur (les processus NextstateLogic et Statelogic):

Le décodeur est commandé par une machine d'état, il y a quatre états dans cette machine d'état, le tableau IV.7 récapitule les états et leur signification.

Nom d'état	signification
SNormal	Le décodeur est prêt, le décodeur met son résultat à la prochaine phase dans le front montant du prochain cycle d'horloge.
SwaitOperand	Le décodeur attend un opérande à partir de la banque de registres.
SReading	Le décodeur attend un opérande mémoire.
SWaitExec	Le décodeur est maintenant prêt mais la prochaine phase (phase d'exécution) n'est pas prête (nous devons l'attendre).

Tableau IV.7 Les états de la machine d'état du décodeur

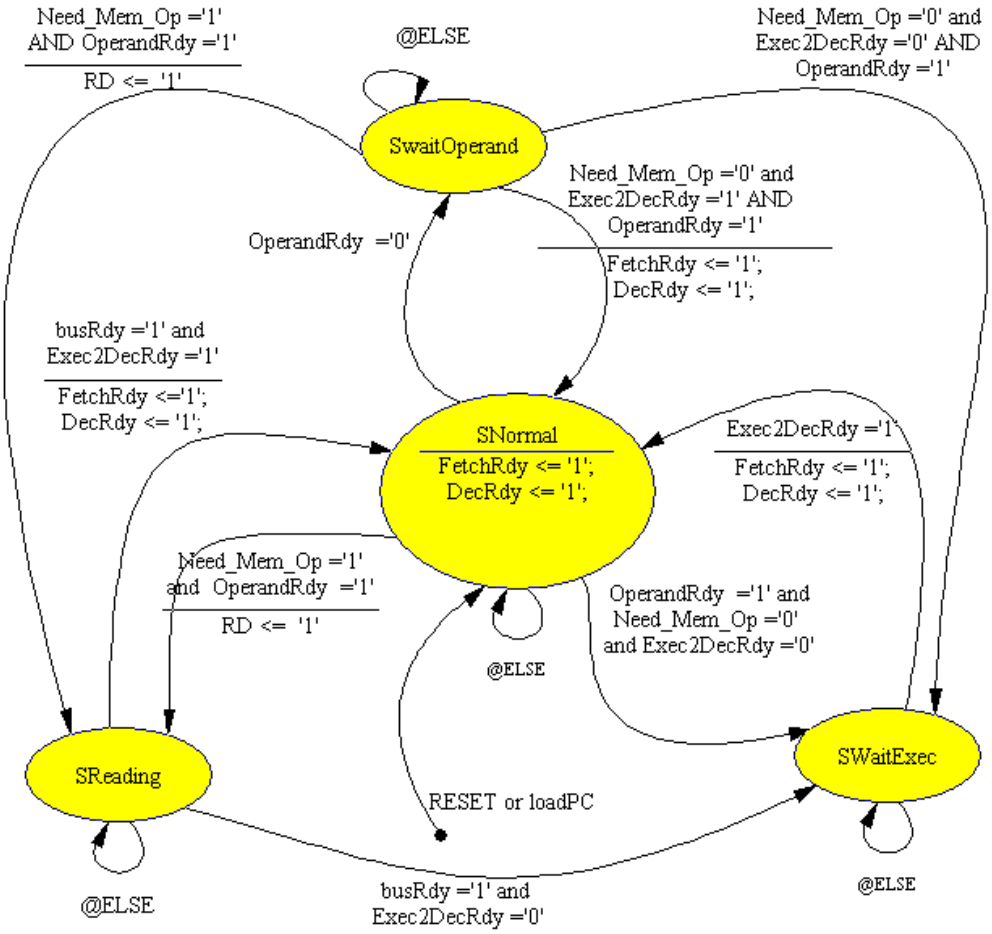


Fig.IV.10 : Le diagramme d'état de la machine d'état du décodeur.

Le processus ***NextstateLogic*** détermine l'état futur (signal **NextState**) à partir de l'état actuel (signal **State**) et les entrées du bloc (signaux **Exec2DecRdy** et **busRdy**) et les signaux

internes (**OperandRdy**, **Need_Mem_Op**). Le processus ***Statelogic*** est la partie séquentielle de la machine d'état, ce processus transfère le signal **NextState** au signal **State** dans le front montant de l'horloge. Les signaux **NextState** et **State** sont utilisé plus tard pour déterminer les valeurs du signal interne **DecRdy** et des signaux de sortie (**RD** et **FetchRdy**), par exemple le signal **DecRdy** (signal prêt de décodeur) est activé quand **NextState** est égal à l'état **SNormal**.

```
429   DecRdy       <=   '1' WHEN NextState = SNormal ELSE   '0';
```

De même, le code VHDL suivant est utilisé pour déterminer le signal de lecture RD (cherchant un opérande mémoire):

```
423   RD           <=   '1' WHEN NextState = SReading ELSE  '0';
```

6. Le processus clockedout:

C'est la partie séquentielle du décodeur. Tous les signaux de sortie allant à la prochaine phase sont enregistré ici (l'instruction décodée est transférée à la phase d'exécution) dans le front montant de l'horloge quand la prochaine phase est prête (**Exec2DecRdy**='1'). Le signal **IDdec** est utilisé ici comme moyen pour différencier entre deux instructions consécutives. Ce signal **IDdec** est basculé (de 0 à 1 ou de 1à 0) chaque front montant de l'horloge quand l'instruction décodée est prête (**DecRdy**='1 ') et la phase d'exécution est prêtte (**Exec2DecRdy**='1 ').

Le registre **IM** (ou le signal **im15to3**) est enregistré (chargé) ici de l'opcode d'instruction (**IR_opcode = LOAD_IM**) ou de la mémoire (l'opérande cherché quand **IR_opcode** égale à **MOV_MEM_REG** et la direction **IR_moveDir** égale à **MEMDIR_READ** et le registre cible est le registre **IM** (**IR_RegGroup = RG_IM**) et quand le bus est prêt (**busRdy = '1 '**).

IV.3.3.5. La phase d'exécution (exec.vhd):

Dans cette phase nous effectuons toutes les opérations arithmétiques et logiques et l'exécution des instructions de saut et d'appel. Les tableaux suivants donnent une description de différents signaux d'entrées et de sortie de cette phase.

Signal	Direction/ largeur	Signification
To_DecRdy	Out/1	L'étage d'exécution est prêt pour une autre instruction décodée
DecRdy	In/1	L'instruction décodée est prête pour l'exécution
Reg_Rx	In/16	Le premier port (premier opérande) de la banque des registres ou le registre SP(R7) en cas des instructions push, pop, ret, et call
ADR_Aoperand	In/16	Le premier opérande de l'instruction décodée
Boperand	In/16	Le deuxième opérande de l'instruction décodée
ALU_Op	In/4	Le code opération d'UAL
DestReg_adr	In /3	Le registre de destination de l'opération
StorToReg	In/1	Demander à la phase d'exécution de sauver le résultat au registre de destinations si ce signal est activé
Condition	In/4	La condition des instructions de branchement conditionnel
ExecVector	In/6	Représente un ensemble de 6 signaux (voir le tableau de la phase décodage)
Mem_DIR	In/1	Direction du transfert de données mémoire (employé pour différencier entre le push et le pop)
PushPopR	In/1	L'instruction décodée utilise le SP pour l'adressage (instructions de pile)
IDdec	In/1	Identification de l'instruction décodée

Tableau IV.8 les signaux d'unité d'exécution reliés à la phase décodeur

Signal	Direction/ largeur	Signification
WB_Rdy	In/1	La phase Write-back est prête pour une autre donnée
WB_Wr	Out/1	la commande d'écriture à la phase Write-back
WB_ADR	Out/16	L'adresse d'écriture mémoire
WB_DATA	Out/16	Les données qui doivent être écrites à la mémoire

Tableau IV.9 les signaux du l'unité d'exécution reliés à la phase Write-Back

Signal	Direction/ largeur	Signification
Custom_Rdy	In/1	Le bloc personnalisé (custom) à un résultat prêt
Cus_Resault	In/16	Le résultat du bloc personnalisé (cutom)

Tableau IV.10 les signaux du l'unité d'exécution reliés au bloc personnalise (custom)

Signal	Direction / largeur	Signification
IDexe	Out/1	Identification de l'instruction exécutée
SP_Wr_En	Out/1	Permission d'écriture du registre SP (active dans les instructions de pile)
SP_ADR	Out/16	La nouvelle valeur de SP (SP +1 ou SP -1)
Reg_Wr_En	Out/1	Permission d'écriture de n'importe quel registre comprenant le SP(R7)
Reg_adr	Out/3	L'adresse de registre (000,001,010…,111)
Reg_data	Out/16	La nouvelle valeur de registre

Tableau IV.11 les signaux du l'unité d'exécution reliés au banque de registres

Signal	Direction/ largeur	Signification
LoadPC	In/1	Commande du chargement de registre PC
NewPC	In/16	La nouvelle adresse de registre PC

Tableau IV.12 les signaux du l'unité d'exécution reliés au bloc fetch

Les signaux du tableau suivant sont internes pour la phase d'exécution.

Signal	Largeur	Signification
INCDEC_out	16	(voir le processus calculp0p1m1)
p0p1m1	16	(voir le processus (calculp0p1m1))
ALU_Dout	16	La sortie de résultat de l'unité UAL
Flags	16	Le registre des drapeaux
IS_Arith	1	Signal booléen qui indique les opérations arithmétiques
IS_condtrue	1	La condition du saut conditionnel est vraie
Must_Write	1	Écrivez les données à la mémoire
Exec_Rdy	1	La phase d'exécution est prête pour une autre instruction décodée

Tableau IV.12 les signaux internes de la phase d'exécution

1. Le processus calculp0p1m1 (l'unité d'incrémentation/décrémentation):

Le but de ce processus est de combiner le matériel utilise pour les instructions arithmétiques (incrémentation et décrémentation) avec le matériel utilise pour les instructions de pile (push,pop et ret) dans une seule composante matérielle, et économisé de ce fait le matériel utilisé. Ce processus place la valeur d'un signal interne **p0p1m1** à une des valeur suivante (0, -1, +1) selon les opérations UAL ou les opérations de la pile (push ou pop), par exemple si l'opération d'UAL est incrémentation d'un registre (**ALU_Op=OP_DEC_RY**) le signal **p0p1m1** doit avoir la valeur -1 (p0p1m1 <= (others=>'1');), et s'il y a une opération de pile (**PushPopR**='1') le signal p0p1m1 prend la valeur de -1 en cas d'écriture mémoire ou empilement (**Mem_DIR=MEMDIR_WRITE**) ou la valeur +1 en cas de lecture mémoire ou dépilement (**Mem_DIR=MEMDIR_READ**).

Après affectation de la valeur correcte au signal **p0p1m1** (0, +1, -1) nous ajoutons ce signal à la valeur du registre du premier opérande **Reg_Rx** (Il s'agit du registre SP en cas des opérations de pile) pour produire le signal **INCDEC_out**. Ce signal est sauvegardé dans la banque de registres en cas d'opération de pile (SP<=SP+/-1) ou employé comme entrée de l'UAL pour être ensuite transmis à l'intérieur de l'UAL vers sa sortie quand l'opération d'UAL est une décrémentation (**ALU_Op = OP_DEC_RY**) ou une incrémentation (**ALU_Op = OP_INC_RY**).

2. Le composant condtest:

Ce composant prend les entrées ; le drapeau zéro **Z**, le drapeau de retenu **C**, le drapeau de signe **S**, le drapeau de débordement **OV** et la condition du branchement **condition** pour produire un signal booléen. Ce signal booléen indique si la condition d'entrée est vraie ou faux. Ce résultat booléen (signal **IS_condtrue**) est employé pour commander l'exécution des instructions de branchement conditionnel (voir la section ci-dessus **"le saut conditionnel"** pour plus de détail).

3. Le composant UAL:

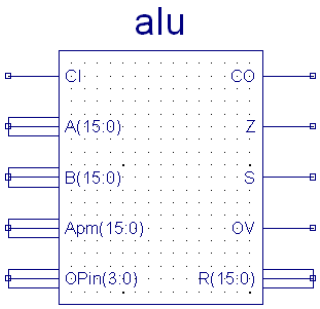


Fig.IV.11 : Le symbole d'UAL

Ce module est un circuit combinatoire pur, ce module effectue toutes les opérations arithmétique et logique. Dans le tableau IV.13: A est le premier opérande des opérations binaires où **B** est le deuxième opérande. Dans les opérations qui ont seulement un opérande, **A** est le seul opérande excepté l'opération **OP_RY** où **B** est le seul opérande. L'opération **OP_RY** est employée pour transférer les données du registre au registre (instruction move Rx,Ry), il y a une autre exception en cas de l'opération **OP_INC_RY**, et l'opération **OP_DEC_RY** où **Apm** est le seul opérande, cet opérande est calculé dans un autre bloc (l'unité d'incrémentation /décrémentation) et envoyer à la sortie de l'unité UAL en cas d'instructions INC ou DEC. Les autres entrées à ce bloc sont la retenue **CI** (utilisé pour les opérations arithmétique) et le code d'opération **OPin**. Les sorties sont ; le résultat **R** de l'opération, et les drapeaux **CO** (sortie retenu), **Z** (drapeau zéro), **S** (drapeau de signe), **OV** (drapeau de débordement), et le signal **arith_Nlogout** (cette sortie indique que des opérations arithmétiques sont exécutées dans l'UAL).

Signal UAL	Direction In/out	Signification
A	in	Premier opérande de 16 bits
B	in	Deuxièmes opérande de 16 bits
Apm	in	La sortie de l'unité d'incrémentation/décrémentation.(A+/-1)
CI	in	La retenue d'entrée
OPin	In	Code d'opération de 4 bits (voir le tableau ci-dessous)
R	Out	Résultat de 16 bits
CO	Out	La retenue de sortie
Z	Out	La sortie Zéro (=1 le résultat est zéro)
S	Out	Le signe (=1 négatif)
OV	Out	Le débordement
arith_Nlogout	Out	Arithmétique ou logique (1= Arithmétique, 0= logique)

Tableau IV.13 les entrées et sorties d'UAL

Constante du code opération VHDL	Opération
OP_NOTRX	Résulta est Not (opérande A)
OP_AND	opération ET entre l'opérande A et B
OP_OR	opération OU entre l'opérande A et B
OP_XOR	opération XOR entre l'opérande A et B
OP_RY	Résulta est opérande B (utilisé dans l'opération de mouvement)
OP_SWPA	Permuter les deux octets de l'opérande A
OP_INC_RY	Incrémenté l'opérande A (le résultat est l'entrée Apm)
OP_DEC_RY	Décrémenté l'opérande A (le résultat est l'entrée Apm)
OP_ADC	L'addition de A et de B avec le retenu d'entrée
OP_ADD	L'addition de A et de B sans retenu d'entrée
OP_SBC	La soustraction de A de B avec le retenu d'entrée
OP_SUB	La soustraction de A de B sans retenu d'entrée
OP_ADDp1	Non implémenter
OP_ADDm1	Non implémenter
OP_TSTC	Mêmes que OP_SBC mais sans sauvegarder le résultat au registre
OP_TST	Mêmes que OP_SUB mais sans sauvegarder le résultat au registre

Tableau IV.14 Les opérations d'UAL

Le calcul des drapeaux sont faits dans le fichier **ALU.vhd** comme suit:

1) Le drapeau de retenu (CO <= sum (N)).
2) Le drapeau de signe est le dernier bit de l'opération arithmétique (S <= sum (N -1)).
3) Le drapeau zéro est placé quand le résultat est zéro (Z<= '1' when sum (N-1 downto 0) ="0000000000000000" else '0').
4) Le drapeau de débordement n'est pas implémenté.

Le tableau IV.14 présente tout les codes opération **OPin** de l'UAL et la fonction correspondante, ce code est identique au champ **operation** dans le format d'instruction **ARITH_LOG**.

4. Le signal prêt d'unité d'exécution (le processus readyp):

Le signal prêt **Exec_Rdy** de l'unité d'exécution est calculé dans le processus ***readyp***. Ce signal est égal au signal prêt **WB_Rdy** de Write-Back quand l'unité d'exécution doit écrire quelques données dans la mémoire (**Must_Write** <= '1';) ou il est égal au signal prêt **Custom_Rdy** du bloc personnalisé CUSTOM quand ce bloc est sélectionnée (instructions personnalisées), dans d'autres cas le signal prêt **Exec_Rdy** est égale à 1 (l'unité d'exécution est prête pour une autre instruction décodée).

5. Le processus clocked:

C'est la partie séquentielle de cette phase. Nous faisons les choses suivantes dans cette phase:

1) Le signal **IDdec** est transféré à **IDexe** chaque fois que l'instruction est exécutée (fini).
2) Calculer l'adresse **WB_ADR** et les données **WB_DATA** et les transférer à la phase Write-Back quand cette phase est prête (**WB_Rdy**='1').
3) Le registre des drapeaux est chargé de l'opérande **B** quand le signal **loadFlag** est activé et l'instruction est de type load ou de la sortie de l'unité UAL quand l'instruction est de type arithmétique (le signal **IS_Arith** est à 1).

6. Autre processus de la phase d'exécution:

Nous commençons par les signaux allant à la banque de registres, le registre de destination de l'opération est l'entrée **DestReg_adr** venant à partir de l'unité de décodeur, ceci est représenté par l'instruction VHDL suivante:

```
179    Reg_adr      <= DestReg_adr;
```

La sortie de données registre **Reg_data** est égale à la sortie **ALU_Dout** d'UAL quand l'instruction personnalisée n'est pas sélectionné (**Custom_Inst** = ‘0’), ou à la sortie du bloc personnalisé **Cus_Resault** quand l'instruction personnalisée est sélectionné:

```
180    Reg_data     <= ALU_Dout when Custom_Inst ='0' else
181                     Cus_Resault ;
```

Le signal de permission d'écriture du port d'écriture normal de la banque de registres **Reg_Wr_En** est le signal décodé **StorToReg** quand l'instruction décodée est prête:

```
183   Reg_Wr_En   <= StorToReg and DecRdy;
```

Le signal de permission d'écriture du port du registre **SP (**le signal **SP_Wr_En)** est le signal décodé **PushPopR** quand l'instruction décodée est prête (**DecRdy**='1 '):

```
184   SP_Wr_En    <= PushPopR  and DecRdy;
```

La nouvelle valeur de SP (**SP_ADR**) est toujours la sortie de l'unité d'incrémentation/ décrémentation **INCDEC_out**.

Le signal **LoadPC** est activé quand l'instruction est décodée (**DecRdy**='1 ') et l'une des conditions suivantes est vrai:

1) des instructions de saut ou d'appel sont exécuté (signal d'entrée **JumpTo**) ou
2) Quand le saut conditionnel (signal d'entrée **Jumpifcond**) est décodé et la condition est vraie (le signal de sortie **IS_condtrue** du composant **condtest**).

Le signal **LoadPC** est activé également quand le signal **Int_Reset** est activé, la remise à zéro est exécutée comme un saut à l'adresse 0.

```
190   LoadPC <= Int_Reset or(((IS_condtrue and Jumpifcond) or JumpTo)and DecRdy);
```

L'adresse de destination de saut NewPC est la sortie **ALU_Dout** d'UAL quand le signal **Reset** n'est pas activé (**Int_Reset**='0 ') ou l'adresse 0(others=>'0 ') quand le signal **Reset** est activé.

```
192   NewPC  <= ALU_Dout when Int_Reset='0'else (others=>'0');
```

IV.3.3.6. Le bloc personnalisé (custom.vhd):

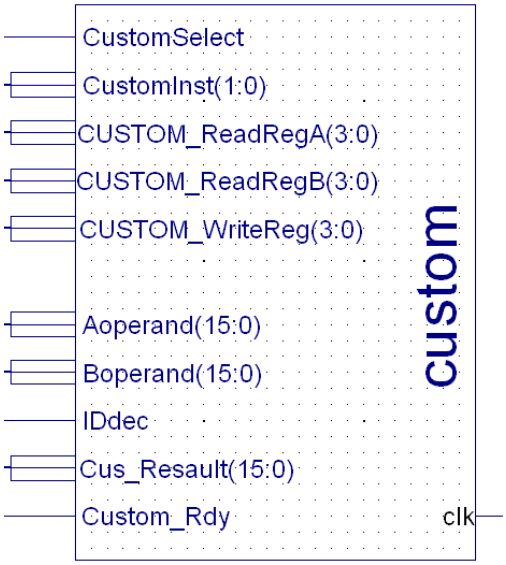


Fig.IV.12 : Le symbole du bloc personnalisé

Ce bloc calcule le résultat des quatre instructions personnalisées. La fonctionnalité de ces instructions n'est pas figée et peut être modifiée dans ce bloc par l'utilisateur final (programmeur). Les entrées de ce bloc viennent de la phase de décodage, et les sorties vont à la phase d'exécution et à la phase de décodage. Ce bloc est sélectionné par l'intermédiaire de l'entrée **CustomSelect** de décodeur et l'instruction à l'intérieur du bloc est choisie par l'intermédiaire de l'entrée **CustomInst** du décodeur. Les entrées de données au bloc sont **Aoperand** et **Boperand**, ces signaux sont de largeur de 16 bits et ils viennent des deux ports de banque de registres et envoyé par le décodeur. Le signal **IDdec** commande le début du calcul, le calcul doit commencer quand ce signal change d'état. Quand le bloc personnalisé a fini l'exécution, le résultat 16 bits doit être placé à la sortie **Cus_Result**, et le signal prêt **Custom_Rdy** doit être activé dès que possible avant le front montant de l'horloge. Le temps d'exécution doit être un multiple entier de la période d'horloge.

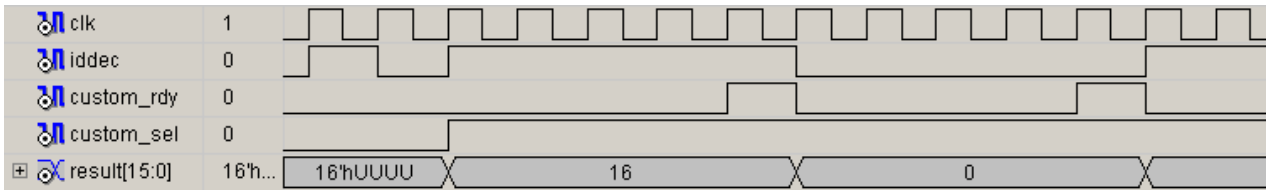


Fig.IV.13 : Le signal prêt du bloc personnalisé (*custom_rdy*) et le début d'exécution du bloc personnalisé. Le calcul commence quand IDdec change d'état et se termine quand le signal prêt devient ` 1'

1. Les besoins d'opérandes de l'instruction personnalisé

Puisque le décodeur étant figée et la personnalisation est seulement faite dans le bloc personnalisé et non pas dans le décodeur, le bloc personnalisé doit indiquer au décodeur que de telles instructions ont besoin du premier opérande **Aoperand** et/ou du deuxième opérande **Boperand** et si de telles instructions doivent stocker le résultat dans un registre de destination ou non. Les signaux **CUSTOM_ReadRegA**, **CUSTOM_ReadRegB**, et **CUSTOM_WriteReg** sont employés à cette fin. Par exemple si **CUSTOM_ReadRegA** = 0100 la troisième instruction (le bit de position 2 égale 1) a besoin de l'opérande **Aoperand** et les autres n'ont pas besoin (les autres positions 0, 1, et 3 égale 0). Le signal **CUSTOM_ReadRegB** est employé pour le deuxième opérande, et le signal **CUSTOM_WriteReg** est utilisé pour indiquer que le premier opérande est employé comme registre de destination ou non.

2. Le calcul du signal prêt

L'utilisateur (programmeur) est responsable du calcul du signal prêt (**custom_rdy**) à l'intérieur du bloc personnalisé. Nous proposons d'employer un compteur chargé par une valeur constante chaque fois que l'exécution du bloc personnalisé commence, cette constante représente le nombre de cycle requis pour le calcul. Le signal prêt est activé quand le compteur atteint une valeur fixe prédéfinie (habituellement "111… 1" ou "000… 0"). Et enfin les valeurs constantes chargées dans le compteur n'ont pas besoin d'être les mêmes pour toute les instructions personnalisé.

IV.4. Conclusion

La modularité est le souci principal de notre conception. L'utilisation du langage VHDL rend notre conception portable. La modularité aide la portabilité de la conception, parceque la division de la conception en blocs et phases ou module réduit au minimum le temps de modification au cas où si un code non-portable est employé dans un certain module. En outre l'utilisation des blocs multiples coupe une machine d'état complexe ou un code complexe en plus petites et plus simples machines d'état ou plus petit et plus simple code. Dans notre travail le code non portable pourrait être situé dans le bloc personnalisé ; les autres blocs sont conçus pour être portable.

Chapitre V: Les Résultats

V.1. Introduction

Le processeur a été complété en deux phases; la phase fonctionnalité de l'architecture de base et la phase fonctionnalité de l'architecture personnalisée ajoutée. Chaque phase a été testée en utilisant les programmes d'instructions que l'on peut voir en Annexe A : *Codes sources et programmes*. Les programmes ont été vérifiés par la simulation. L'outil principalement utilisé est le *Xilinx ISE 7.1i Project Navigator* pour la simulation, la Synthèse et l'implémentation.

V.2. La vérification du processeur de base

Quand le code VHDL d'une conception est complété, l'étape suivante serait de créer les tests de référence 'testbenchs' pour simuler la conception. Ces tests peuvent être sous différentes formes (VHDL, Verilog, SystemC...). Le testbench encapsule la conception, ce qui leurs permettent d'injecter des stimulis dans la conception sous test, et contrôler la sortie de la conception. Si les formes d'onde de sortie de la conception ne sont pas comme prévues, un "bug" se produit. Quand un "bug" est trouvé, le concepteur doit "debugger" les formes d'ondes de sortie et décider si le "bug" vient de la conception ou du testbench. Dans les deux cas, le bug doit être corrigé et la simulation est à effectuer de nouveau. Quand la forme d'onde de sortie de la conception est comme prévue le concepteur peut passer à la phase suivante du flot de conception (la synthèse).

V.2.1. Le schéma du processeur

Dans ce qui suit nous donnerons le schéma du processeur utilisant les différents blocs et phases. Le bloc de RAM n'étant pas une partie du processeur, il contient le programme que le processeur doit exécuter. En utilisant les marqueurs d'entrée/sortie de l'éditeur de schémas de Xilinx, on peut tester et debugger les signaux internes du processeur. L'utilisation d'un bloc de mémoire pour les instructions et les données dans ce schéma indique l'adoption de l'architecture de Von Neumann dans la conception du processeur. En ajoutant une mémoire séparée d'instruction et en rebranchant les blocs du processeur l'architecture de Harvard peut être vérifiée.

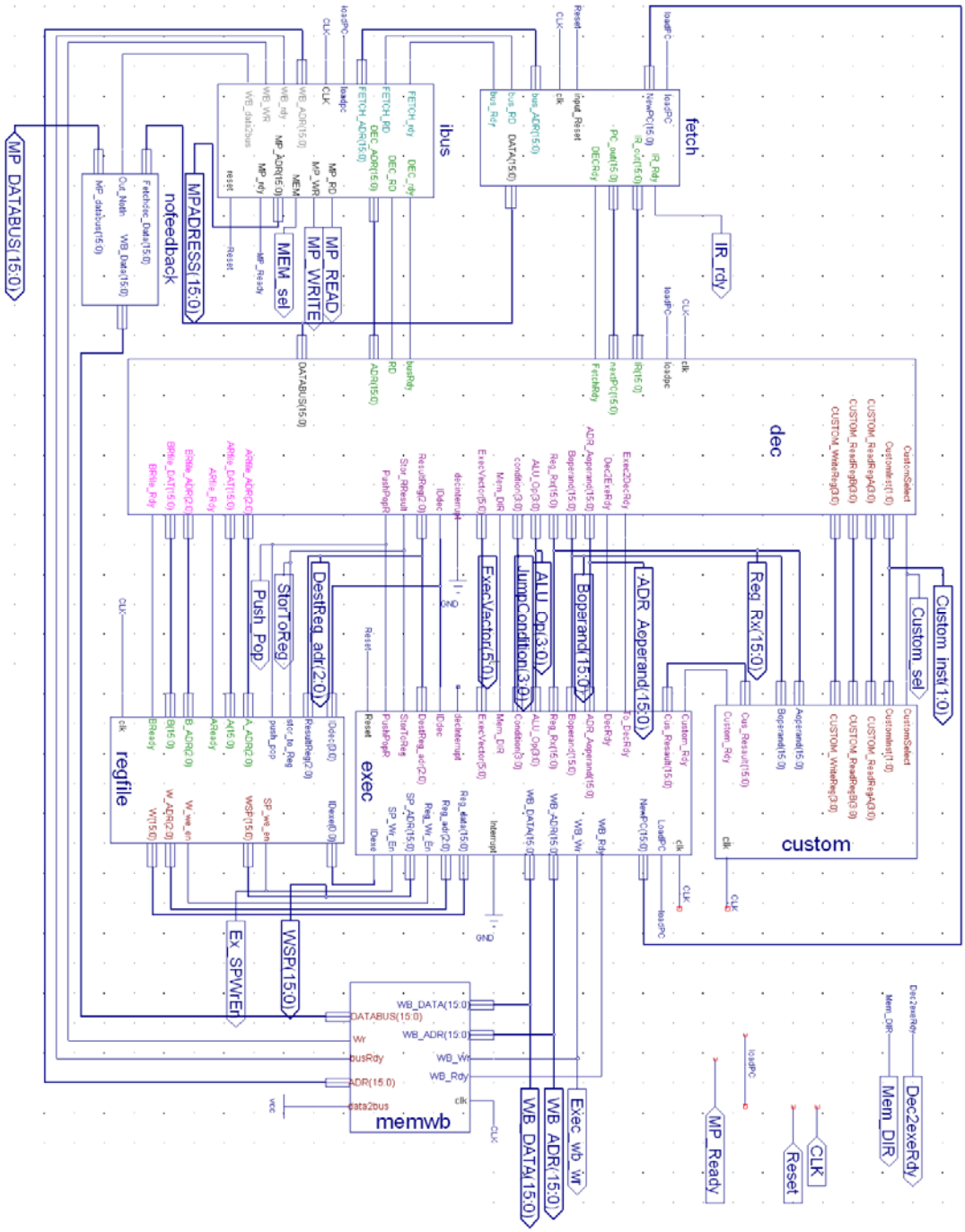


Fig.V.1 : Schéma du processeur

V.2.2. Le test incrémentiel

Puisque nous avons besoin d'un nombre important de vecteurs de test pour examiner tous les blocs et toutes les phases séparément, nous utilisons une technique que nous appelons le test

incrémental. Dans cette technique nous testons le premier bloc seul puis nous connectons un deuxième bloc avec le premier bloc et nous testons les deux blocs ensemble et ensuite nous connectons un troisième bloc et ainsi de suite. Puisque les signaux entre les blocs n'ont pas besoin d'être produits extérieurement (du testbench), les vecteurs de test sont réduits au minimum. La réduction des vecteurs de test est importante parceque ceux-ci réduisent les erreurs venant du testbench et minimisent le temps nécessaire pour créer ces vecteurs.

Dans notre processeur, le premier bloc évalué est **ibus** ensuite on ajoute le bloc de recherche d'instruction puis le décodeur et le banque de registres ensuite le bloc d'exécution et enfin le bloc Write-Back. (Voir l'Annexe B : *les Simulations*).

V.2.3. Le testbench et le débogage d'un signal interne

Le débogage d'un signal interne est important pour vérifier la fonctionnalité de la conception. Nous ajoutons des marqueurs d'entrée/sortie dans le schéma du processeur pour examiner ces signaux. Par exemple le signal **decrdy** est un signal interne pour le processeur qui est utilisé exclusivement pour des fins de débogage.

Après avoir ajouté des marqueurs d'entrée /sortie pour les signaux importants nous employons un testbench de forme d'onde pour créer des stimulis pour notre processeur. Les seuls signaux qui sont des entrées (reçoivent ainsi les stimulis du testbench) au processeur sont les CLK, RESET, et INTERRUPT, tout les autres signaux sont des sorties.

V.2.4. Le programme de test

Tester les fonctionnalités du processeur revient à tester le processeur quant il exécute un programme, nous utilisons un programme qui contient différents types d'instructions (move, dec, inc, push, call, ret, pop, and jump). Ce programme est stocké en mémoire, le tableau suivant (Tableau V.1) montre le contenu de cette mémoire.

Ce programme et d'autres sont déclarés comme une constante VHDL **Memory_Content** dans le package **programs** (voir l'annexe A : *Codes sources et programmes*).

Adresse	Instructions	Signification
0	a_loadim(10)	IM =10
1	a_move(REGx_1, 5)	R1 =85 (=IM*8 + 5 =10*8+5)
2	a_loadim(32)	IM =32
3	a_move(REGx_7, 0)	SP =256 (= IM*8+0)
4	a_loadim(1)	IM =1
5	a_move(REGx_2, 0)	R2 =8 (= IM*8+0)
6	a_dec (REGx_2,REGx_2)	R2 = R2 - 1
7	a_inc (REGx_1,REGx_1)	R1 = R1 +1
8	a_pushR(REGx_1)	Push R1 to the stack
9	a_loadim(8)	IM=8
10	a_call (0)	call address = 64 (=IM*8+0)
11	a_popR (REGx_3)	Pop R3 from the stack (the value of R1 when pushed in 8).
12	a_loadim(0)	IM=0
13	a_jmp (6)	jump address = 6 (=IM*8+6)
The following is a subroutine called from address 10		
64	a_rjmp (1)	Relative Jump to next address (pc+1)+1 = 66
65	a_move (REGx_5,REGx_1)	R5 <- R1
66	a_Ret	Return from subroutine.

Tableau V.1 Le programme de test

V.2.5. La simulation fonctionnelle

Les figures suivantes donnent le résultat de la simulation fonctionnelle quand le processeur exécute le programme ci-dessus, dans la figure V.2 nous observons que la phase de recherche fonctionne comme prévu, par exemple le bus de données **databus** a la valeur de 26279 avec son adresse (**memadr** + 1) avant le temps 162.5 ns apparaît comme une sortie de la phase de recherche (signal **IR**) après le temps 162.5 ns. L'effet du signal **Reset** doit initialiser l'adresse de recherche ainsi le registre **PC** à 0 et abandonner n'importe quel travail effectué dans n'importe quelle phase (excepté la phase du writeback), l'initialisation de l'adresse de recherche est montrée au temps 512.5ns comme sortie du bus d'adresses du processeur **memadr**.

Dans la figure V.3 nous observons que le pipeline du processeur fonctionne comme nous prévoyons, sur cette figure l'instruction push R1 (opcode 23680) est cherchée en mémoire au temps 1325ns, et après deux cycles d'horloge (temps 1425ns) cette instruction apparait comme une sortie de la phase de recherche (signaux **IR**, et **PCOUT**), à ce moment (temps=1425ns) le décodeur cherche les opérandes de cette instruction (le registre **R1**, et le registre **SP**) et commence à décoder cette instruction, un cycle plus tard (temps = 1475ns) les opérandes de cette instruction (signal **Boperand**, signal **Reg_Rx**) apparaissent comme des entrées de la phase

d'exécution (Sortie du décodeur) avec les différents signaux décodés (le signal **pushpopr** et d'autres). Dans la phase d'exécution, une fois l'instruction décodée est détectée, on procède à:

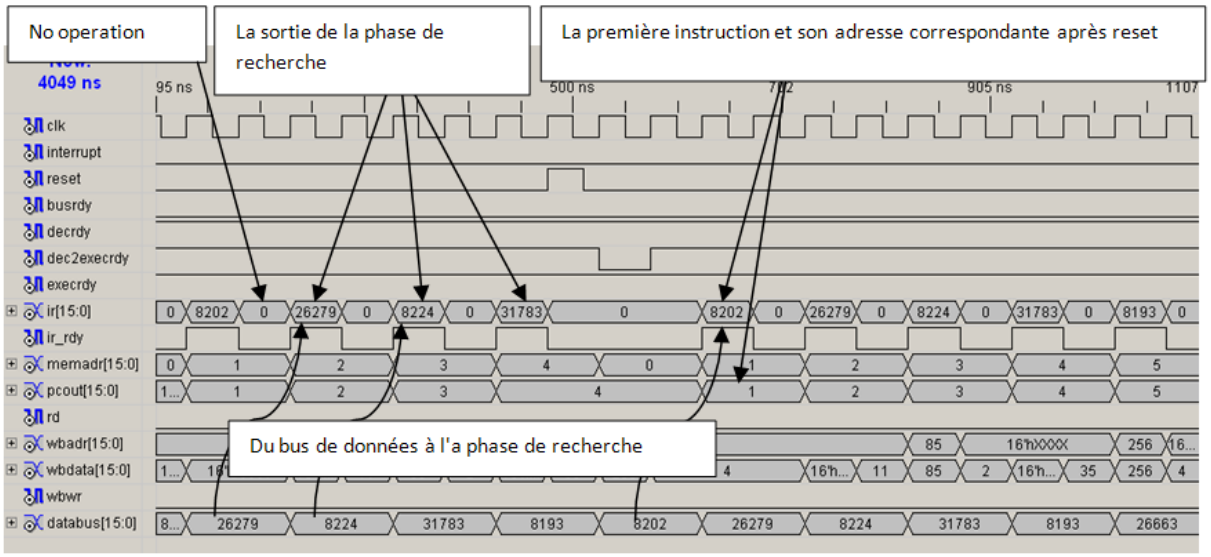


Fig.V.2 : La phase de chercher d'instructions et l'opcode d'instructions

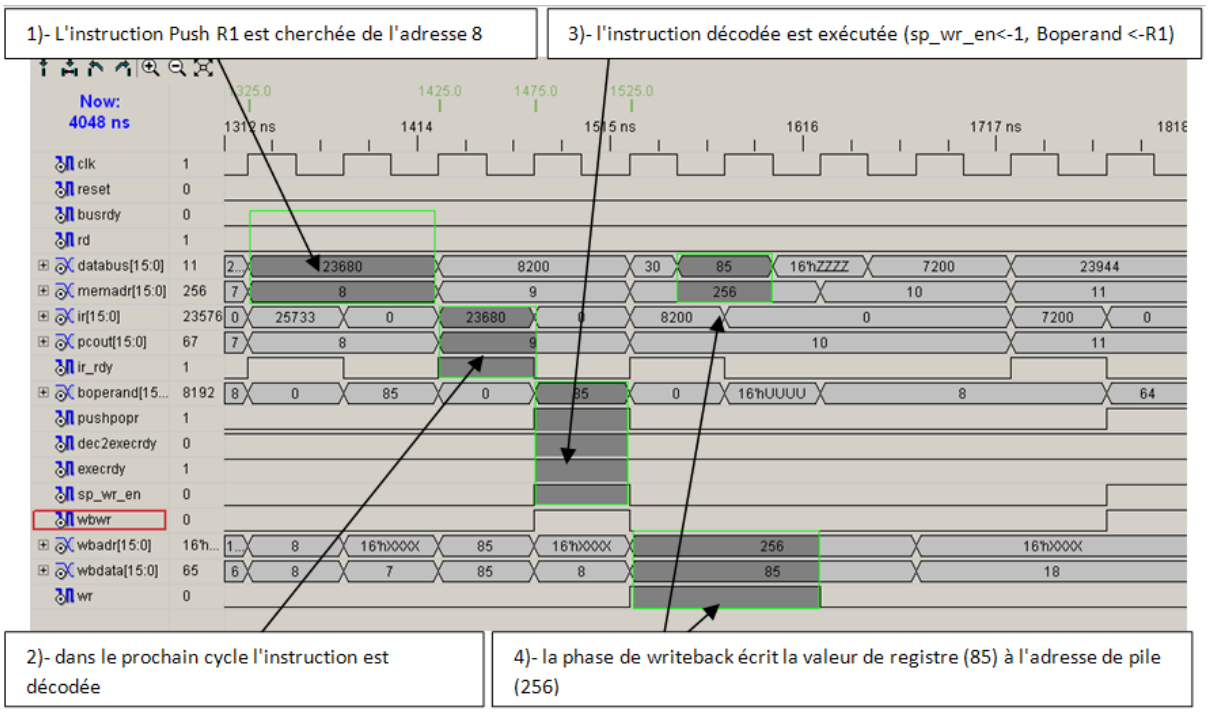


Fig.V.3 : Le pipeline du processeur

1) Incrémenter le registre **SP** (signal **Reg_Rx**).
2) Activer le signal **sp_wr_en** (permission d'écriture de registre **SP**) pour écrire la nouvelle valeur du **SP** dans le banque de registres.

3) Mettre la valeur du registre R1 (signal **Boperand**) et son adresse (**SP**) à la phase writeback, et activer le signal de commande **wbwr** pour écrire la valeur de **R1** en mémoire (Pile).

Enfin la phase du writeback commence le cycle d'écriture au temps 1525ns.

V.2.6. Autre méthode de débogage et l'utilisation du code non synthétisable

L'utilisation de la forme d'onde est une bonne méthode quand la conception est petite ou le temps de simulation n'est pas trop long, par exemple si le temps d'exécution du programme est une seconde et l'horloge est de 100 mégahertz, alors il y a un million de transitions entre 0 et 1 de presque tous les signaux, la vérification d'une telle conception prend beaucoup de temps. Pour surmonter ce problème, nous ajoutons un code de débogage VHDL dans chaque fichier de conception, par exemple le code suivant (figure V.4) est ajouté au fichier **regfile.vhd**. Ce code génère un message quand des données sont écrites dans le banque de registres, dans ce code, le premier test d'IF est exécuté quand le **SP** et autre registre sont écrits en même temps (**SP_we_en = '1' and W_we_en = '1'**), ceci se produit quand l'instruction POP est exécutée; le deuxième test est exécuté quand le **SP** est le seul registre écrit (**W_we_en = '0 ' and SP_we_en = '1'**), ceci se produit dans l'instruction de retour (ret); le troisième test est exécuté quand un registre normal est écrit et la pile n'est pas concerne (**W_we_en = '1 'et SP_we_en = '0'**), ceci se produit dans les instructions de mouvement ou d'UAL.

```
104  -- pragma translate off
105  if DEBUG_REGFILE  then
106
107    if  SP_we_en   ='1' and W_we_en ='1' then
108          ASSERT false report "Wr pop SP="& integer'image(co_integer(WSP))
109                              &" Reg("& integer'image(WSel)&")="& integer'image(co_integer(W));
110    elsif  SP_we_en   ='1'then
111          ASSERT false report "Wr Push SP="& integer'image(co_integer(WSP));
112    elsif  W_we_en ='1' then
113          ASSERT false report "Wr Reg("& integer'image(WSel)&")="& integer'image(co_integer(W));
114    end if;
115
116  end if;
117  -- pragma translate_on
```

Fig.V.4 : Code VHDL ajouté pour des fins de debugage dans regfile.vhd

Le code ci-dessus nous aide à trouver beaucoup de problèmes fonctionnels, la figure suivante (figure V.5) est la sortie de la fenêtre de simulation du code VHDL ci-dessus quand nous exécutons la simulation du processeur avec le problème de l'unité incrémentation/décrémentation (figure V.5.a) et après nous résolvons ce problème (figure V.5.b).

```
at 250.000 ns(1): Wr Reg(1) = 85
at 330.000 ns(1): Wr Reg(7) = 256
at 410.000 ns(1): Wr Reg(2) = 8
at 450.000 ns(1): Wr Reg(2) = 8
at 490.000 ns(1): Wr Reg(1) = 85
at 530.000 ns(1): Wr Push SP=256
at 650.000 ns(1): Wr Push SP=256
at 870.000 ns(1): Wr Push SP=256
at 970.000 ns(1): Wr pop SP=256 Reg(3)=11
at 1.150 us(1)  : Wr Reg(2) = 8
at 1.190 us(1)  : Wr Reg(1) = 85
```

(a) Problème du push/pop.

```
at 250.000 ns (1): Wr Reg(1) = 85
at 330.000 ns(1): Wr Reg(7) = 256
at 410.000 ns(1): Wr Reg(2) = 8
at 450.000 ns(1): Wr Reg(2) = 7
at 490.000 ns(1): Wr Reg(1) = 86
at 530.000 ns(1): Wr Push SP=255
at 650.000 ns(1): Wr Push SP=256
at 870.000 ns(1): Wr Push SP=257
at 970.000 ns(1): Wr pop SP=258 Reg(3)=40
at 1.150 us(1)  : Wr Reg(2) = 6
at 1.190 us(1)  : Wr Reg(1) = 87
```

(b) problème du push/pop résolu.

Fig.V.5 : la sortie de la fenêtre de simulation et le problème de push/pop

Dans la figure ci-dessus nous voyons que les instructions push et pop ne modifient pas le registre SP. Par inspection, on peut constater que le problème était dans l'unité d'incrémentation/décrémentation. Après correction du problème, nous simulons la conception une fois encore et nous remarquons que le problème est corrigé (figure V.5.b).

En général, il est convenable d'imprimer un message dans la fenêtre de simulation ou d'écrire des informations dans un fichier pendant la simulation fonctionnelle. Dans ce cas, ce code n'est pas synthétisable. Pendant la synthèse, ces lignes de code doivent être rejetées. Ceci peut être fait en plaçant une ligne spéciale devant et après le code non synthétisable. La ligne suivante est placée devant le code non synthétisable:

```
65   -- pragma translate_off
```

Et la ligne suivante est placée après le code non synthétisable:

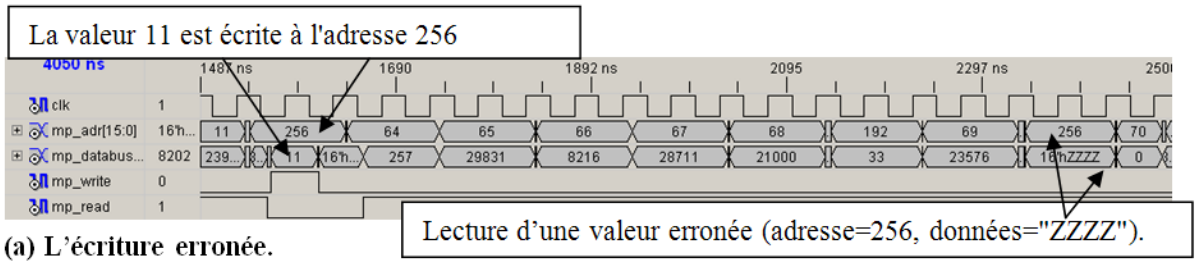
```
78   -- pragma translate_on
```

L'inconvénient d'employer un code non synthétisable est l'incompatibilité entre la simulation fonctionnelle et la simulation temporelle, ainsi elle est non recommandée. En revanche, l'utilisation du code non synthétisable dans le testbench ne crée pas d'incompatibilité entre la simulation fonctionnelle et la simulation temporelle si nous utilisons le même testbench pour les deux simulations.

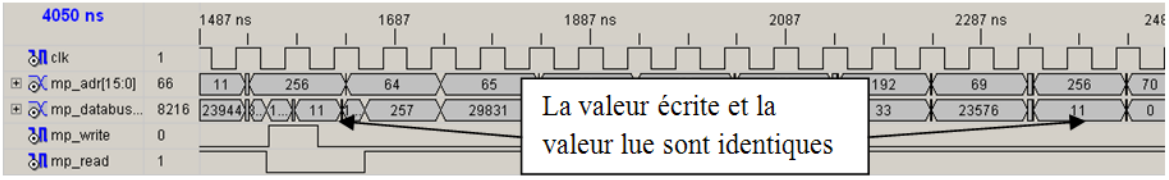
V.2.7. La simulation temporelle

Puisque le bloc RAM ne fait pas toujours partie de l'FPGA et son code n'est pas nécessairement synthétisable (nous essayons de synthétiser notre bloc RAM mais la synthèse a échoué après 4 heures), nous aurons déplacé le bloc de RAM à l'intérieur du testbench pour vérifier le processeur en utilisant la simulation temporelle. Le diagramme du bloc pour le testbanch et le processeur est donné ci-dessous (figure V.7).

La simulation temporelle utilise les informations de retard des blocs et de l'intercommunication de l'outil de routage pour donner une évaluation plus précise du comportement de la conception. La simulation temporelle (ou post-place and route simulation) est une partie fortement recommandée du flot de conception HDL. La simulation temporelle permet la simulation qui étroitement correspond à l'opération réelle du dispositif. Effectuer une simulation temporelle aidera à découvrir les problèmes qui ne peuvent pas être trouvés dans la simulation fonctionnelle, par exemple nous découvrons le problème de données qui ne sont pas correctement écrites en mémoire (voir la figure V.6.a et b). Noter que le problème n'est pas découvert dans la simulation fonctionnelle. La source de problème est que le signal de commande écrire du processeur **mp_write** et le bus de données changent de valeurs en même temps. Le problème est résolu par le retardement du signal de sortie de permission d'écriture de bus de données de la phase writeback **WB_data2bus**, ce signal est généré dans le bloc **ibus.**



(a) L'écriture erronée.



(b) l'écriture résolu.

Fig.V.6 : La simulation temporelle du processeur avec le problème d'écriture erronée(a) et la résolution du problème (b)

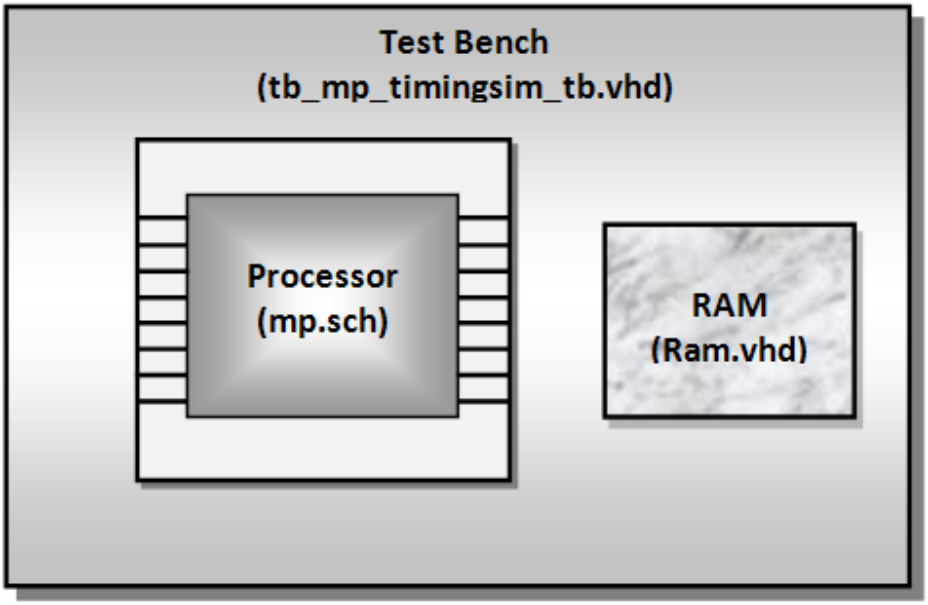


Fig.V.7 : Le diagramme du bloc de testbanch et le processeur

La simulation temporelle dans son intégralité est donnée dans l'annexe B : *les Simulations*.

V.2.8. L'architecture de Harvard

Une technique fondamentale pour améliorer la performance de recherche d'instructions est de concevoir un processeur avec deux interfaces mémoire, une pour les instructions et l'autre pour les données. Ceci est désigné sous le nom de l'architecture de Harvard, par rapport à une architecture conventionnelle de Von Neumann dans laquelle les instructions et les données partagent la même mémoire. Dans l'architecture de Harvard, les recherches d'instructions ne sont pas perturbées par les accès des autres blocs. Malheureusement, l'architecture de Harvard présente de nombreux problèmes au niveau système par exemple, la façon de diviser la mémoire de programme et la mémoire de données, et comment charger des programmes en mémoire à laquelle on ne peut pas accéder par des opérations de type charger/stocker. Les processeurs modernes utilisent cette architecture localement en utilisant des caches séparées pour les instructions et les données [33]. Notre processeur est rebranché d'une manière à tirer profit de cette architecture. Le tableau V.2 montre le temps d'exécution et l'utilisation du bus du même programme (calcul inverse de matrices 4x4) utilisant des architectures de Harvard et de Von Neumann. Dans le tableau, la période d'horloge du processeur est de 20ns (50 mégahertz) et le cycle du bus est de 40ns.

Architecture	Harvard	Von Neumann
Temps d'exécution	**59,98 us**	**78,30 us**
Nombre de cycles d'instruction	1487	1486
Nombre de cycles de lecture des données (décodeur)	293	293
Nombre de cycles d'écriture des données (writeback)	174	174
Utilisation du bus d'instruction	99.1%	99.8% (instruction et données)
Utilisation du bus de données	31.1%	-

Tableau V.2 Comparaison entre les architectures Harvard et Von Neumann de processeur

L'architecture de Harvard est préférée quand le bus de données est trop utilisé; un exemple de ceci est les applications de traitement d'image. L'architecture de Von Neumann est préférée quand l'algorithme n'accède pas souvent en mémoire (applications de commande) ou la minimisation du coût est d'une importance primordiale.

V.3. La vérification du Processeur avec instructions personnalisée

Pour vérifier les instructions personnalisées et ainsi le bloc personnalisé nous écrirons un programme de test et nous choisissons quatre opérations (les instructions personnalisées) programmées en langage VHDL dans le fichier **custom.vhd**. Dans ce qui suit (figure V.8), on donne le programme de test (le fichier **program_customtest.vhd**) utilisé pour vérifier les instructions personnalisées.

```
16              (0=>  a_loadim( 10 ),
17               1=>  a_move(REGx_1, 5) ,          --  reg1 =10*8+5  =85
18               2=>  a_loadim( 32 ) ,
19               3=>  a_move(REGx_7, 0) ,          --  SP  = 32*8+0  =256
20               4=>  a_loadim( 1 ) ,
21               5=>  a_move(REGx_2, 0) ,          --  reg2  = 1*8+0  =8
22               6=>  a_move(REGx_3, REGx_1) ,
23               7=>  a_move(REGx_4, REGx_2),
24
25               8=>  a_Custom1 (REGx_1,REGx_2) ,
26               9=>  a_Custom2 (REGx_3,REGx_4) ,
```

Fig.V.8 : Le programme de test (fichier program_customtest.vhd)

Dans la première partie (de l'adresse 0 à 7) les registres R1, R2, R3, R4 et SP sont initialisés, et dans la dernière partie (l'adresse 8 et 9) deux instructions personnalisées sont appelées avec les opérandes de registre appropriés. Nous implémentons le comportement de chaque instruction personnalisée dans le fichier **custom.vhd** et les signaux **result1**, **result2**, **result3**, et **result4** pernnent le résultat de chacune d'elles. Par exemple result2 est le résultat de

l'instruction personnalisée **a_custom2** et est calculé pour donner le maximum des deux opérandes comme indiqué à la figure V.9.

```
121   calcul_inst1: process(Aoperand , Boperand)        -- max
122     begin
123
124       Resault1 <= Aoperand;
125
126       if Aoperand < Boperand then
127
128        Resault1 <= Boperand;
129
130       end if;
131
132     end process;
```

Fig.V.9 : Calcul du result1 de l'instruction a_custom2 (custom.vhd)

Le résultat final du bloc personnalisé est choisi parmi les résultats intermédiaires dans la déclaration (instruction VHDL) finale selon le signal d'entrée de sélection d'instruction **CustomInst** (à partir du décodeur).

```
153     WITH CustomInst SELECT
154     Cus_Resault <= Resault0                  WHEN "00",
155                    Resault1                  WHEN "01",
156                    Resault2                  WHEN "10",
157                    Resault3                  WHEN "11",
158                    (Cus_Resault'range=>'-') WHEN OTHERS ;
```

Fig.V.10 : Choix de résultat (custom.vhd)

La figure suivante (figure V.11) montre la forme d'onde de simulation du code dans les figures V.8, V.9, et V.10.

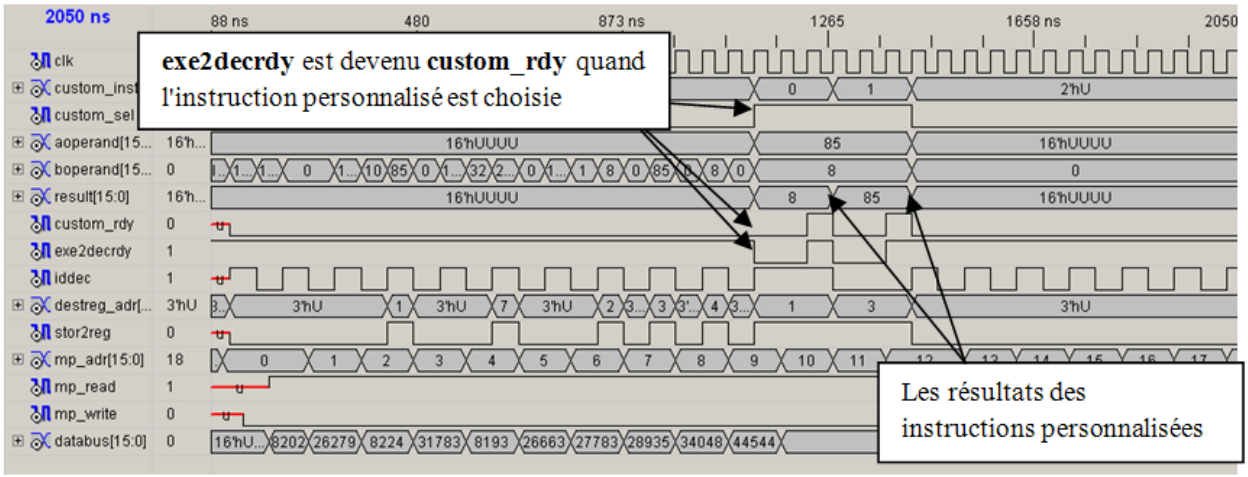


Fig.V.11 : Résultats de simulation du processeur avec les instructions personnalisées

La performance d'instructions personnalisée et le nombre de cycles mémoire

Le deuxième test des instructions personnalisées est une comparaison entre un programme dans lequel on emploie des instructions personnalisées et un programme qui emploie des instructions normales. A la figure V.12 on donne les deux programmes, chacun calcule la même fonction (la valeur minimum d'un tableau de nombres entiers).

Le premier programme (figure V.12 a) emploie les instructions personnalisées et le deuxième programme emploie seulement les instructions normales (figure V.12 b). Dans les deux programmes R3 conserve l'adresse de début du tableau (256) et R4 l'adresse de fin (256+9).

```
17  0=>  a_loadim( 256/8 ),
18  1=>  a_move(REGx_3, 0) ,   -- R3 =256 first address
19  2=>  a_loadim( 256/8 +1 ) ,
20  3=>  a_move(REGx_4, 1) ,        -- R4 = 256 + 9
21
22
23  4=> a_loadReg(REGx_3, REGx_1 ), -- R1 = MEM[R3]
24  5=>  a_inc  (REGx_3,REGx_3) ,   -- R3 = R3 + 1
25  6=> a_loadReg(REGx_3, REGx_2 ), -- R2 = MEM[R3]
26
27  7=>  a_Custom1 (REGx_1,REGx_2), -- R1 = min(R1,R2)
28  8=>  a_test    (REGx_3,REGx_4),
29
30  9=> a_jmpNEq   ( -5 ),        -- PC+1 -5 =9+1-5 =5
31
32
33
34
```

(a) Avec instructions personnalisées

```
17  0=>  a_loadim( 256/8 ),
18  1=>  a_move(REGx_3, 0) ,          -- R3 = 256
19  2=>  a_loadim( 256/8 +1 ) ,
20  3=>  a_move(REGx_4, 1) ,          -- R4 = 256 + 9
21
22
23  4=> a_loadReg(REGx_3, REGx_1 ),   -- R1 = MEM[R3]
24  5=>  a_inc  (REGx_3,REGx_3) ,     -- R3 = R3 + 1
25  6=> a_loadReg(REGx_3, REGx_2 ),   -- R2 = MEM[R3]
26
27
28  7=>  a_test    (REGx_1,REGx_2),       --
29  8=>  a_jmpLs   ( 1 ),                 --   a_rjmp
30  9=>  a_move(REGx_1, REGx_2) ,         --
31
32  10=>  a_test      (REGx_3,REGx_4),
33
34  11=> a_jmpNEq   ( 256-7 ),  -- PC+1 -7  =11+1-7 =5
```

(b) Sans instructions personnalisées

Fig.V.12 : Programme avec instructions personnalisées(a) et sans instructions personnalisée(b)

Quand nous examinons les deux programmes, il est évident que l'utilisation des instructions personnalisées réduit la taille des programmes. Dans notre programme nous économisons deux mots dans la mémoire réservée au programme. Ainsi, l'utilisation des instructions personnalisées réduit le temps d'exécution et les accès mémoire. Dans le tableau suivant on donne le temps d'exécution et le nombre d'accès mémoire pour les deux programmes.

Programme	avec des instructions personnalisées exécutées dans		Sans instructions personnalisées
	un cycle d'horloge	3 cycles d'horloge	
temps d'exécution	6895 ns	7025 ns	8741 ns
Nombre des accès mémoire	68	68	87

Tableau V.3 Comparaison entre programme avec instructions personnalisés et programme sans instructions personnalisées

Le tableau V.3 montre que le temps d'exécution et l'accès mémoire sont sauvés malgré que les instructions personnalisées soient exécutées en trois cycles d'horloge.

V.4. La vérification de la reconfiguration

Normalement la vérification est faite dans un circuit réel avec une reconfiguration partielle FPGA activée. La simulation de la reconfiguration partielle est difficile ou elle n'est pas possible du tout. La reconfiguration des instructions personnalisées signifie la reconfiguration du bloc personnalisé. Dans notre stratégie de vérification nous choisissons un algorithme avec deux parties et nous développons deux blocs personnalisés (**custom1.vhd, custom2.vhd**), les deux blocs correspondent aux instructions personnalisées choisies pour chaque partie de l'algorithme. Les deux blocs sont intégrés et multiplexés dans le bloc **custom.vhd** et le signal **Cus_Config** choisit un des deux blocs, la reconfiguration est simulée en changeant ce signal à 0 quand **custom1.vhd** est configuré, et à 1 quand **custom2.vhd** est configuré. Le dispositif de reconfiguration qui pilote la reconfiguration du signal **Cus_Config** est modelée d'une façon comportementale (pas nécessairement synthétisable) dans le bloc **config_device.vhd**. Le dispositif de reconfiguration est commandé par une routine software, quand on a besoin de la reconfiguration on appelle cette routine, dans notre test cette routine est appelée entre les deux parties de l'algorithme. Enfin le bloc **desplay2sim.vhd** est ajouté pour afficher les résultats dans la fenêtre de simulation. L'interconnexion entre les blocs est donnée dans le schéma suivant (figure V.13).

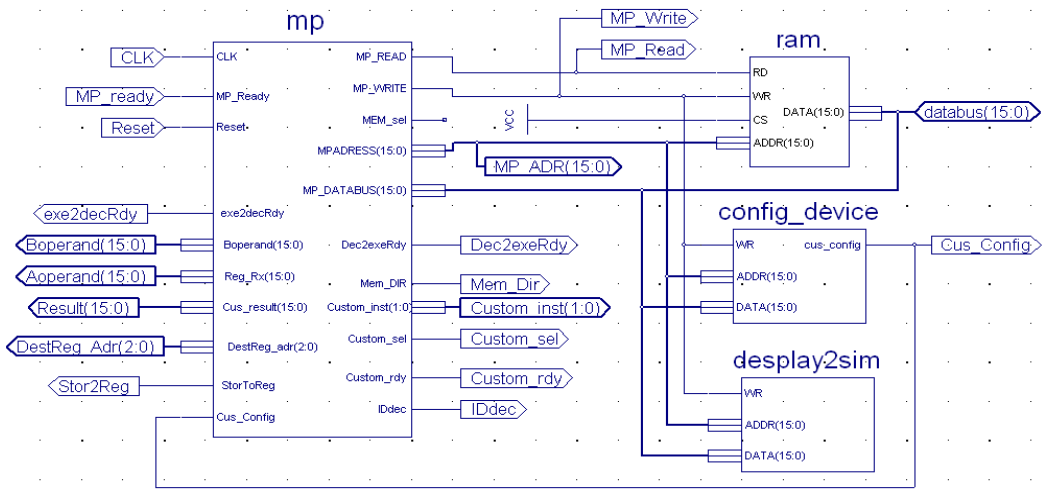


Fig.V.13 : Schéma de reconfiguration du processeur

V.4.1. L'algorithme

L'algorithme calcule le déterminant et l'inverse d'une matrice carrée. Pour simplifier le développement du bloc personnalisé nous utilisons le format à point fixe plutôt que le format à point flottant pour représenter les nombres réel. Dans le format à point fixe [34] le point décimal est fixé dans une seule position. Dans notre programme de test les nombres à point fixe sont de largeur de 16 bits et le point décimal est fixé au milieu (7ème bit – 8ème bit). Par exemple le nombre 1.0 est représenté par '00000001 00000000' dans notre format à point fixe.

L'algorithme (en code C) est donné à la figure V.14. L'algorithme est testé en utilisant le code C ci-dessus (en utilisant les nombres flottants) puis le code est transformé en un code intermédiaire, le code intermédiaire utilise des nombres à point fixe (classe fixed en C++) et des constructions simples en C/C++ (aucune des boucles *for* ou *while*, par example la boucle *for* est éliminée par l'utilisation de *if* et *goto*), ensuite le code est transformé en assembleur (notre assembleur). La figure V.14 montre la transformation d'un fragment de code. Le programme complet transformé **program_invmatReconfigCustom.vhd** est donné en annexe A : *Codes sources et Programmes*.

```
// first part use custom1.vhd
// the input matrix A is concatenated with
//the identity matrix I to form AI matrix = [A I]
// make the matrix triangular (superior) using
//lines transformation (line(i) = line(i) + M * line(j) ).
for(int i=0;i<Matrix_Size -1 ;i++)
    { float  AIii=  1 / AI[i][i];              // custom instruction 1
      for(int L=i+1;L<Matrix_Size;L++)
           { float  M=  AI[L][i] *AIii ;        // custom instruction 2
             for(int j=0;j<Matrix_Size*2;j++)
                 AI[L][j] =   AI[L][j] - M* AI[i][j];// custom instruction 3
           }
    }
// make the matrix diagonal using lines transformation
for(int i=Matrix_Size-1;i>0 ;i--)
    { float  AIii=  1 / AI[i][i];
      for(int L=i-1;L>=0;L--)
         { float M=  AI[L][i] * AIii ;
           for(int j=0;j<Matrix_Size*2;j++)
                AI[L][j] =   AI[L][j] - M* AI[i][j];
         }
    }
// we call to the routine that configure the custom2.vhd in this point
//calculate the determinant of a resultant diagonal matrix
Float  Det=1;
for(int i=0;i<Matrix_Size ;i++)   Det = Det * AI[i][i] ;//custom instruction 1
printf("\ndet: %f \n",Det);
//make the matrix equal to  [ I Ainv ] using line
//transformation (line(i) = M * line(i) )
for(int i=0;i<Matrix_Size ;i++)
    { float M =  AI[i][i];
      for(int j=0;j<Matrix_Size*2;j++ )
           AI[i][j] =  AI[i][j]  / M;   //custom instruction 2
    }
```

Fig.V.14 : L'algorithme avec les instructions personnalisées dans les parties 1 et 2 de l'algorithme

```
                                fixed *R1=  &AI[0][0];          90=> a_loadim(     MATRIX_ADR /  8 ),
                                                                91=>a_move(REGx_1, MATRIX_ADR mod 8) ,
float det=1;                    fixed R2=1;                     92=>a_loadim(     One /  8 ),
for(int i=0;i<Matrix_Size ;i++)                                 93=>a_move(REGx_2, One mod 8) ,
 {                                                              94=>a_loadim(     MATRIX_SIZE /  8 ),
                                int R4= MATRIX_SIZE;            95=>a_move(REGx_3, MATRIX_SIZE mod 8) ,
                                loopd:                          --loopd:
  det = det * AI[i][i] ;        fixed  R5= *R1;                 96=>a_loadReg(REGx_1, REGx_5 ),
                                R2 = R2 * R5 ;                  97=>a_Custom2(REGx_2,REGx_5) ,
                                                                98=>a_loadim(      (AI_MATRIX_SIZE + 1 ) /  8 ),
                                R1 += Matrix_Size*2 +1 ;        99=>a_add   (REGx_1, (AI_MATRIX_SIZE +1) mod 8),
                                R4--;                           100=>a_dec   (REGx_3) ,
}                               if (R4) goto loopd;             101=>a_jmpNEq       ( - 6 ),
```

Fig.V.15 : La transformations du code C

V.4.2. Les instructions personnalisées

Dans la figure ci-dessus (figure V.14) nous choisissons trois fragments de code pour exécuter comme des instructions personnalisées dans la première partie de l'algorithme et comme deux fragments de code dans la deuxième partie. Les tableaux suivants montrent les instructions et les opérations correspondantes programmées dans **custom1.vhd** et **custom2.vhd**.

Instructions personnalisées (custom1.vhd)	Operation à point fixe (avec operande registre)	Le code correspondant en langage C
A_custom1	1 / Ra	1 / AI[i][i]
A_custom2	Ra * Rb	AI[L][i] *AIii
A_custom3	Ra - M* Rb	AI[L][j] - M* AI[i][j]

Tableau V.4 Les instructions personnalisées de la première partie du programme

Dans la troisième instruction personnalisée **a_custom3** la valeur **M** est sauvée dans le bloc personnalisée quand nous exécutons la deuxième instruction **a_custom2** et ainsi nous n'avons pas besoin d'un troisième registre (noter bien que notre jeu d'instructions est basé sur deux opérandes registre).

Instruction personalisée (custom2.vhd)	Operation à point fixe (avec operande registre)	Le code correspondant en langage C
A_custom1	Ra*Rb	Det * AI[i][i]
A_custom2	Ra / Rb	AI[i][j] / M

Tableau V.5 Les instructions personnalisées de la deuxième partie du programme

Dans le deuxième tableau (Tableau V.5), l'instruction **A_custom1** (Ra*Rb) pourrait être remplacée par **A_custom2** de la première partie, et **A_custom2** pourrait être remplacée par

A_custom4 de la première partie et la reconfiguration n'est donc pas nécessaire. Le code global n'est pas optimisé, mais nous utilisons ce choix pour démontrer le processus de reconfiguration. En général, n'importe quel algorithme qui pourrait être divisé en parties, alors ces parties pourrait être traitées et optimisées séparément dans notre méthodologie.

V.4.3. Résultats de simulation et discussion

Le programme est testé en utilisant des matrices NxN (N = 2, 3, 4, 5) et les résultats sont les même que les résultats du code C original, la figure V.16 est le résultat de la simulation en utilisant une matrice 4x4. Le temps d'exécution de la première partie est environ 71098 ns et l'algorithme (première et deuxièmes parties) est de 90340ns sans comptabiliser le temps de reconfiguration.

Pour montrer la différence dans le temps d'exécution entre un programme qui utilise la reconfiguration (programme reconfigurable) et un autre programme qui reste dans la configuration de démarrage, nous développons un autre programme **program_invmatNoconfigCustom.vhd** qui utilise seulement **custom1** dans les deux parties. Le tableau suivant donne le temps d'exécution des deux programmes utilisant des matrices carrées 2x2, 3x3, 4x4, et 5x5.

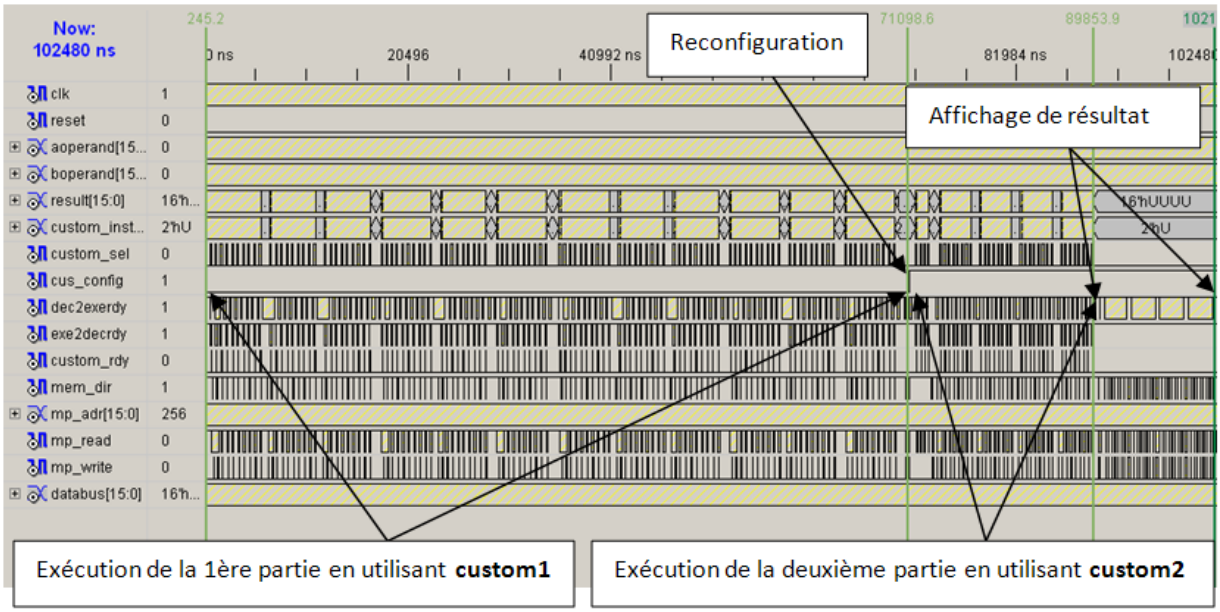


Fig.V.16 : La simulation fonctionnelle de la reconfiguration du processeur (programme inverse de matrice)

Taille de la matrice	2x2	3x3	4x4	5x5
Programme Reconfigurable (µs)	15,58	42,06	90,30	167,02
Programme statique (µs)	16,14	42,90	91,42	168,42
Différence (µs)	0,56	0.84	1.12	1.40

Tableau V.6. Comparaison entre un programme reconfigurable et un programme statique

Le tableau montre que les différences entre les deux programmes sont minimes. La raison de ceci est que le temps d'exécution de la deuxième partie représente une petite fraction du temps d'exécution total. La deuxième raison est que l'instruction **A_custom1** de la deuxième partie est utilisé comme **A_custom2** de la première partie dans le programme statique c'est ainsi que la vitesse est réduite.

V.5. Les résultats d'implémentation à l' FPGA

V.5.1. L'entrée de la conception

1. L'affectation manuelle des pins de la conception

Xilinx recommande [21] que nous devons laisser au programme du placement et du routage de définir les pins de la conception. L'affectation des pins peut parfois dégrader la performance du résultat de l'outil de placement et du routage. Cependant, dans certain cas il peut être nécessaire de désigner les pins de conception de sorte qu'elles puissent être intégrées dans une carte électronique (PCB). Puisque nous ne sommes pas liés à une carte spécifique, nous suivons la recommandation de Xilinx et laissons l'outil de placement et du routage définit les pins de la conception.

2. L'utilisation des modules spécifiques au matériel

Puisque notre processeur est portable, il peut être utilisé dans tous les FPGA ou dans des cartes basées sur FPGA. Mais l'utilisation du matériel spécifique à une FPGA à l'intérieur du bloc personnalisé comme les mémoires embarquées (embedded RAM) ou les multiplicateurs embarqués rend le processeur non portable et nous pourrons être ramené à opérer certaines modifications pour l'employer dans d'autre FPGA non compatible.

V.5.2. La synthèse

La synthèse fournit un lien entre un langage HDL et une netlist, de la même qu'un compilateur C fournit un lien entre le code C et le langage machine. Une fois qu'un modèle HDL est complété deux éléments sont requis pour procéder : un synthétiseur et une bibliothèque de cellules de l'FPGA cible (target library). Le code HDL est mappé aux cellules de cette bibliothèque. Le sommaire de l'utilisation du dispositif (device utilization summary) pour une implémentation des deux FPGA de notre processeur est donné dans le Tableau V.6.

les ressources	Spartan3 (xc3s200)		Virtex4 (xc4vfx100)	
	Utilisation	Pourcentage	Utilisation	Pourcentage
Number of Slices	447/1920	**23%**	487/42176	**1%**
Number of Flip Flops	259/3840	6%	265/84352	0%
Number of 4 input LUTs	763/3840	19%	783/84352	0%
Number of bonded IOBs	38 / 173	21%	38 / 668	5%
Number of GCLKs	1 / 8	12%	1 / 32	3%

Tableau V.7. Le pourcentage d'utilisation du dispositif FPGA de notre processeur (extrait du rapport de l'outil de synthèse)

Ce tableau montre que notre processeur utilise seulement 23 % de la surface totale (les Slices) d'un ancien dispositif (Xilinx Spartan3 xc3s200). Avec un dispositif plus récent (Virtex4), le processeur à utilise seulement qu'1% de la surface totale. Ce tableau est extrait du rapport de synthèse quand nous utilisons le fichier de conception **mp_synthesis** et les dispositifs FPGA comme entrée.

V.5.3. La fréquence maximale

En diminuant l'entrée de la contrainte de la période d'horloge, nous obtenons la période d'horloge réelle comme sortie de l'outil de placement et du routage. Quand l'outil de placement et du routage n'arrive pas à satisfaire la contrainte, nous obtenons dans ce cas-ci la période d'horloge minimale et par conséquent la fréquence maximale de la conception. Dans notre implémentation la période minimale est de **16.31ns** (fréquence maximale **de 61 mégahertz)** quand nous employons le FPGA *Spartan3* (le dispositif xc3s200) et la période minimale est de **6.084ns** (fréquence maximale de 164 **mégahertz)** quand nous employons le *Virtex4* FPGA (dispositif xc4vfx100). Les deux tableaux V.8 et V.9 donnent les résultats d'implémentation de notre conception dans les deux dispositifs (*Spartan3* et *Virtex4*) employant la contrainte de période

d'horloge comme entrée et la période d'horloge réelle résultante. Les données des tableaux sont extraites des rapports de l'outil de placement et du routage.

Constraint (the clock is 50% HIGH and 50% LOW)	Requested	Actual	Logic Levels
TS_CLK =PERIOD TIMEGRP"CLK" 25 ns	25.000ns	22.90ns	6
TS_CLK =PERIOD TIMEGRP"CLK" 20 ns	20.000ns	17.33ns	5
TS_CLK =PERIOD TIMEGRP"CLK" 17 ns	17.000ns	16.50ns	6
*TS_CLK=PERIOD TIMEGRP"CLK" 16 ns	16.000ns	**16.31ns**	7

Tableau V.8. Les périodes d'horloge demandée et réelle de la conception implémentée dans FPGA *Spartan3*

Constraint (the clock is 50% HIGH and 50% LOW)	Requested	Actual	Logic Levels
TS_CLK =PERIOD TIMEGRP"CLK" 16 ns	16.000ns	8.488ns	4
TS_CLK =PERIOD TIMEGRP"CLK" 8 ns	8.000ns	7.393ns	7
TS_CLK =PERIOD TIMEGRP"CLK" 7 ns	7.000ns	6.982ns	5
TS_CLK =PERIOD TIMEGRP"CLK" 6.5 ns	6.500ns	6.466ns	5
*TS_CLK=PERIOD TIMEGRP"CLK" 6 ns	6.000ns	**6.084ns**	6

Tableau V.9. Les périodes d'horloge demandée et réelle de la conception implémentée dans FPGA *Virtex*

V.6. Conclusion

Les simulations fonctionnelles et temporales sont deux méthodes de vérification; elles sont bonnes pour détecter un nombre importent d'erreurs fonctionnelles et temporales. Le test incrémental ajoute à ces méthodes la possibilité de diminuer le nombre de signaux qui doit être vérifié dans les conceptions qui ont des blocs (conception modulaire). L'utilisation du code non synthétisable aide seulement dans la simulation fonctionnelle, ainsi il est conseillé d'utiliser au début de la conception. L'utilisation des instructions personnalisées peut accélérer les applications et réduire le nombre d'instructions dans l'application et ainsi le nombre de cycles mémoire. La réduction du nombre d'instructions réduit l'effort de conception et ainsi le temps de conception. Les résultats de ce chapitre montrent que rendre les instructions personnalisées variable (reconfigurable) peut diminuer le temps d'exécution de l'application. Dans le calcul du temps de l'exécution, nous n'incluons pas le temps nécessaire pour modifier l'FPGA (ou le temps de reconfiguration) quand on bascule d'une ensemble d'instructions personnalisées à une autre ensemble d'instructions personnalisées. Le temps de reconfiguration diminue de génération en génération et devient ainsi moins important dans des futur FPGA. Finalement, Les résultats d'implémentation FPGA montrent que notre processeur a seulement besoin de 1% de la totalité de la surface des FPGA modern, et peut tourner avec une vitesse d'horloge de 164 MHz malgré l'utilisation des constructions VHDL portable dans la conception du processeur.

Conclusion générale

Conclusion générale

Les objectifs de cette thèse ont été atteints avec la construction d'un nouveau processeur 16 bit de type pipeline qui supporte les instructions variable (personnalisées).le processeur implémente 25 instructions dans un pipeline en cinq phases qui utilise l'architecture classique de Von Neumann pour la mémoire et qui peut être modifiée à l'architecture de Harvard. Le jeu d'instruction de base a été choisi de telle façon de fournir et compléter la fonctionnalité de base du processeur. Les instructions personnalisées sont conçues par l'utilisateur final pour être utilisées dans différents applications.

L'implémentation a démontré qu'elle pouvait fonctionner à la vitesse maximale de 164 MHz dans l'FPGA Virtex sans utiliser d'instanciations spécifiques (ou composants). Le processeur a été vérifie dans des simulations tant fonctionnelle que temporelle. Le processeur de base (ou le jeu d'instructions de base) ne fonctionne qu'avec des entiers 16 bits, mais les instructions personnalisées ou variables pourraient fonctionner avec des données ayants d'autres type et formats.

Le simulateur de jeu d'instructions n'est pas utilisé dans notre travail. Généralement, ce simulateur est utilisé dans l'étape de conception d'un nouveau processeur lorsque ce dernier n'est pas encore implémenté. La principale caractéristique de ce simulateur est la vitesse. En effet, il permit de simuler le démarrage d'un système d'exploitation dans un délai raisonnable.

Le processeur n'a pas été testé dans une FPGA réelle et le flot de reconfiguration partiel na pas été utilisé, en d'autres termes, la vérification en circuit n'est pas effectuée.

La vitesse du processeur pourrait être augmentée avec un routage efficace. D'autres recherches sont nécessaires pour confirmer cette idée que le processeur pourrait être amélioré en définissant manuellement les chemines de routage au lieu de permettre à l'outil de tout router automatiquement.

Les tampons des instructions recherchées pourraient facilement être remplaces avec un cache d'instructions dans de futures implémentations

Bibliographie

Bibliographie

[1] **Katherine Compton and Scott Hauck**
An Introduction to Reconfigurable Computing
Invited Paper, IEEE Computer, April, 2000.

[2] **Yung-Chuan Jiang and Jhing-Fa Wang**
Temporal Partitioning Data Flow Graphs for Dynamically Reconfigurable Computing
IEEE transactions on VLSI systems, vol. 15, no. 12, december 2007

[3] **Scott Hauck and Andr´e DeHon**
Reconfigurable Computing The Theory And Practice Of FPGA-Based Computation
Copyright © 2008 by Elsevier Inc.

[4] **M.D. Galanis, G. Theodoridis, S. Tragoudas, D. Soudris, and C.E. Goutis**
Mapping Computational Intensive Applications to a New Coarse-Grained Reconfigurable Data-Path
pp. 652–661, 2004. © Springer-Verlag Berlin Heidelberg 2004

[5] **Paolo Bonzini1, Dilek Harmanci2 and Laura Pozzi1**
A Study of Energy Saving in Customizable Processors
Springer-Verlag Berlin Heidelberg 2007 pp. 304–312, 2007.

[6] **Stefan Raaijmakers and Stephan Wong**
Run-Time Partial Reconfiguration For Removal, Placement And Routing On The Virtex-Ii Pro 1-4244-1060-6/07/ 2007 IEEE.

[7] **michalis d. galanis and gregory dimitroulakos costas e. goutis**
Performance Improvements from Partitioning Applications to FPGA Hardware in Embedded SoCs The Journal of Supercomputing, 35, 185–199, 2006

[8] **JASON VILLARREAL and DINESHSURESH**
Improving Software Performance with Configurable Logic Design Automation for Embedded Systems, 7, 325-339, 2002. 2002 KluwerAcademic Publishers

[9] **Marcela Zuluaga, Theo Kluter, Philip Brisk, Nigel Topham, Paolo Ienne**
Introducing Control-Flow Inclusion to Support Pipelining in Custom Instruction Set Extensions
978-1-4244-4938- 2009 IEEE

[10] **Huynh Phung Huynh and Tulika Mitra**
Runtime Adaptive Extensible Embedded Processors—A Survey
K. Bertels et al. (Eds.): SAMOS 2009, LNCS 5657, pp. 214–224, 2009. c_Springer-Verlag Berlin Heidelberg 2009

[11] **FelixM uhlbauer and Christophe Bobda**
A Dynamic Reconfigurable Hardware/Software Architecture for Object Tracking in Video Streams
EURASIP Journal on Embedded Systems Volume 2006, Article ID 82564, Pages 1–8

[12] **David Caliga and David Peter Barker**
Delivering Acceleration: The Potential for Increased HPC Application Performance Using Reconfigurable Logic

Proceedings of the ACM/IEEE SC2001 Conference (SC'01) 1-58113-293-X/01 © 2001

[13] **Ludovic Devaux, Daniel Chillet, Sebastien Pillement, Didier Demigny**
Flexible Communication Support For Dynamically Reconfigurable FPGAs
978-1-4244-3846-4/09/2009 IEEE

[14] **Michael J. Wirthlin and Brad L. Hutchings**
A Dynamic Instruction Set Computer
1995 IEEE

[15] **Grant McFarland**
Microprocessor Design A Practical Guide from Design Planning to Manufacturing
Copyright © 2006 by the McGraw-Hill Publishing Companies, Inc.

[16] **Steve Kilts**
Advanced FPGA Design Architecture, Implementation, and Optimization
2007 by John Wiley & Sons

[17] **Digital ASIC Group**
Digital ASIC Design A Tutorial on the Design Flow
October 20, 2005

[18] **Hassan Hassan and Mohab Anis**
Low-Power Design of Nanometer FPGAs: Architecture and EDA
Copyright © 2010, Elsevier Inc.

[19] **Michael John Sebastian Smith,**
Application-Specific Integrated Circuits
Addison-Wesley and Benjamin Cummings C 1997

[20] **PONG P. CHU**
RTL Hardware Design Using Vhdl Coding For Efficiency, Portability, And Scalability
Copyright 2006 by John Wiley & Sons, Inc.

[21] **Xilinx ISE software documentation**
Copyright 2005xilinx inc.

[22] **Clive "Max" maxfield**
Design warrior's guide to FPGAs
2004, mentor graphics corporation and Xilinx,inc

[23] **Stamatis Vassiliadis and Dimitrios Soudris**
Fine- and Coarse-Grain Reconfigurable Computing
2007 Springer

[24] **Jari Nurmi**
Processor Design System-on-Chip Computing for ASICs and FPGAs
2007 Springer

[25] **Sivarama P. Dandamudi**
Guide to RISC Processors for Programmers and Engineers
2005 Springer Science+Business Media, Inc.

[26] **Dimitris Mandalidis, Panagiotis Kenterlis, John N. Ellinas**

A Computer Architecture Educational System based on a 32-bit RISC Processor
Copyright © 2008 Praise Worthy Prize

[27] **David Sheldon, Rakesh Kumar, Roman Lysecky, Frank Vahid, Dean Tullsen**
Application-Specific Customization of Parameterized FPGA Soft-Core Processors
ICCAD'06, November 5-9, 2006, San Jose, CA Copyright 2006 ACM 1-59593-389-1/06/0011

[28] **Masayuki Hiromoto, Shin'ichi Kouyama, Hiroyuki Ochi, and Yukihiro Nakamura**
A Retargetable Compiler for Cell-Array-Based Self-Reconfigurable Architecture
IJCSNS International Journal of Computer Science and Network Security, VOL.7 No.4, April 2007

[29] **Etienne Bergeron, Marc Feeley, and Jean Pierre David**
Hardware JIT Compilation for Off-the-Shelf Dynamically Reconfigurable FPGAs
Springer-Verlag Berlin Heidelberg 2008 pp. 178–192, 2008.

[30] **Weisheng CHONG, Sho Ogata, Masanori HARIYAMA and Michitaka KAMEYAMA**
Architecture of a Multi-Context FPGA Using Reconfigurable Context Memory
Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) 1530-2075/05

[31] **MichaelJ.WirthlinandBradL.Hutchings**
A Dynamic Instruction Set Computer
IEEE Workshop on FPGAs for Custom Computing Machines, Napa, CA, April,1995.

[32] **Meyer, Andreas.**
Principles of functional verification
Copyright _ 2003, Elsevier Science

[33] **Mark Balch**
Complete Digital Design
Copyright © 2003 by The McGraw-Hill Companies, Inc.

[34] **JEAN-PIERRE DESCHAMPS, GERY JEAN ANTOINE BIOUL and GUSTAVO D. SUTTER**
Synthesis Of Arithmetic Circuits Fpga, Asic, And Embedded Systems
Copyright # 2006 by John Wiley & Sons, Inc.