UNIVERSITÉ DU QUÉBEC

MÉMOIRE PÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE

DE LA MAITRISE EN PHYSIQUE

PAR

CLAUDE EDDY RABEL

# RÉALISATION D'UN PROCESSEUR RISC POUR LA LOGIQUE FLOUE

16 SEPTEMBRE 1995

# REMERCIEMENTS

La réalisation de ce mémoire n'aurait jamais vu le jour sans la collaboration et le concours de mon directeur, M. Jacob Davidson. Ses précieux conseils m'ont aidé pendant toute la durée de ce travail. À cet effet, je lui adresse mes plus sincères remerciements.

Je remercie également mes frères Romane et Hector Rabel ainsi que mes parents pour leur support moral et financier.

Finalement, je remercie particulièrement Yvelie Phanord, en qui j'ai trouvé une source inépuisable d'inspirations, un support moral continu et beaucoup d'encouragements.

# RÉSUMÉ

Ce mémoire présente l'architecture d'un nouveau processeur flou qui comporte un mécanisme décisionnel en logique floue, avec un nombre de règles programmable, permettant de prendre des décisions adéquates en temps réel. Le processeur flou utilise la technique de consultation de tableau, la composition "max-min" ainsi que la méthode du centre de gravité.

Ce mémoire présente également l'architecture d'un nouveau processeur RISC ainsi que les différents aspects de sa conception. Le processeur RISC contient 81 registres et il utilise 24 instructions et cinq modes d'adressage. La description du processeur a été réalisée avec le langage de description de matériel VHDL et sa mise en oeuvre a été effectuée avec des circuits FPGAs de la famille XILINX-4000.

Le jeu d'instructions du processeur est optimisé pour les applications décrites avec la logique floue. Cette approche a conduit à la réalisation d'un processeur RISC spécialisé pour la logique floue.

# TABLES DES MATIÈRES

# LISTE DES FIGURES

# LISTE DES TABLEAUX

**Tableau** **Page**

# LISTE DES ABRÉVIATIONS

| | |
|---|---|
| CAO | Conception Assistée par Ordinateurs |
| CISC | «*Complex Instruction Set Computer*» |
| CLB | Cellules Logiques Configurables |
| COG | «*Center Of Gravity*» |
| FPGA | «*Field Programmable Gate Array*» |
| IEEE | «*Institut of Electrical and Electronics Engineers*» |
| » | |
| IOB | «*Input Output Block*» |
| IR | «*Instruction Register*» |
| PC | «*Program Counter*» |
| PTF | Pointeur de Fenêtres |
| RAM | «*Random Access Memory*» |
| RISC | «*Reduced Instruction Set Computer*» |
| SCR | «*Status and Control Register*» |
| UAL | Unité Arithmétique et Logique |
| VHDL | «*Very high speed integrated circuits Hardware Description Language*» |
| VLSI | «*Very Large Scale Integrated systems*» |

# LISTE DES SYMBOLES

| | |
|---|---|
| $t$ | temps d'exécution |
| $\mu$ | fonction d'appartenance |
| $\forall$ | pour tout élément |
| $\Re$ | relation floue |
| $\circ$ | composition de |
| $\subset$ | inclus |
| $\subseteq$ | strictement inclus |
| $\not\subset$ | n'est pas inclus |
| $\leq$ | inférieur ou égal |
| $\geq$ | supérieur ou égal |
| $\cup$ | Union |
| $\cap$ | Intersection |
| $\varnothing$ | Ensemble vide |

# INTRODUCTION

La réalisation de processeurs est une tâche passionnante qui, à première vue, peut paraître très compliquée puisqu'elle peut nécessiter des connaissances dans des domaines très variés tels que : la physique, l'informatique et la micro-électronique.

Ordinairement, pour effectuer l'étude des phénomènes naturels ainsi que des processus artificiels, les systèmes informatiques conventionnels utilisent un modèle mathématique qui, dans certains cas, est difficile à réaliser ou à utiliser. Cette difficulté peut être attribuée à des procédés trop complexes ou mals connus, à des calculs mathématiques inutilisables en temps réel, ou encore, à des modèles mathématiques trop compliqués.

Il arrive couramment que l'on dispose d'informations vagues, approximatives ou incertaines qui ne peuvent pas être décrites de manière satisfaisante par la logique conventionnelle. Ces situations sont le résultat d'une certaine méconnaissance d'un événement ou d'une réalité dont la validité n'est pas claire, rigoureuse ou précise. L'être humain est particulièrement habile à comprendre et à traiter ces imprécisions, ces incertitudes et ces approximations en utilisant des raisonnements très simples.

Puisque la logique floue peut représenter de telles informations, les systèmes flous modélisent le raisonnement humain au lieu d'un algorithme mathématique. Dans ces systèmes, on utilise souvent un processeur spécialisé conçu pour la logique floue. Cette approche permet d'obtenir des processeurs très rapides mais elle impose un nombre fixe ou limité de règles floues, d'entrées et de sorties. Ces processeurs sont utilisés pour un nombre restreint d'applications et, de plus, ils ont besoin d'un processeur hôte qui leur fournit les données à analyser. Le nombre de bits représentant

une valeur floue et la méthode de défuzzification sont aussi fixes. Afin d'obtenir une meilleure flexibilité fonctionnelle, on a développé un processeur RISC ( «*Reduced Instruction Set Computer*») spécialisé pour la logique floue. Actuellement, cette logique est utilisée avec succès dans plusieurs domaines tels que l'économie, les systèmes experts, la robotique, l'intelligence artificielle, le traitement d'image et l'automatisme.

Dès le début du développement des systèmes informatiques, les concepteurs ont eu le besoin d'implanter des tâches de plus en plus complexes et performantes. Grâce à l'évolution technologique qui a permis l'intégration d'un plus grand nombre de transistors, la complexité des systèmes a pu être augmentée très rapidement. De plus, le temps d'exécution est abaissé à l'aide de technologies plus rapides. Ainsi, comme la puissance d'un processeur peut se définir par le temps $(t)$ requis pour exécuter une tâche, les systèmes intégrés sont devenus de plus en plus performants, d'où :

$$t = NIE \times NCI \times DC \ , \tag{1}$$

c'est-à-dire que le temps $(t)$ dépend du nombre d'instructions exécutées $(NIE)$, du nombre de cycle par instructions $(NCI)$ et la durée du cycle $(DC)$.

Afin de diminuer davantage le temps $(t)$ et d'augmenter l'efficacité d'un système, les concepteurs ont cherché à réduire le nombre d'instructions nécessaires à accomplir un travail. À cause de cela, on a créé des instructions très compliquées pouvant accomplir le travail de plusieurs instructions simples. Cette méthode favorise des assembleurs riches en instructions et en modes d'adressage. Les processeurs qui utilisent cette méthodologie architecturale sont appelés CISC ( «*Complexe Instructions Set Computer*») ou ordinateurs à jeu d'instructions complexes. Les instructions complexes augmentent souvent la taille et la complexité d'un processeur. Elles sont aussi encombrantes, difficiles à matérialiser et très coûteuses en terme de performance. De plus, plusieurs d'entre elles sont redondantes ou très peu utilisées.

Il est important de souligner que, pendant longtemps, la lenteur des circuits à mémoires par rapport aux processeurs ralentit l'exécution à la lecture des instructions. En ayant des instructions plus complexes, on a pu exécuter une tâche avec un minimum

d'instructions et limiter alors les accès en mémoire. Ceci a permis de réduire le temps ($t$) et d'obtenir de meilleures performances, d'autant plus que la limitation de la capacité de la mémoire et l'importance accrue des programmes (systèmes d'exploitations, base de données, etc.) ont nécessité cette approche.

Durant ces vingt dernières années, la croissance des puces à mémoires a été remarquable. Elles ont passé d'une capacité de 10 Kbits en 1976 à 16 Mbits en 1992 pour une augmentation moyenne de capacité d'environ 50% par an et elles ont connu aussi une progression très significative de leur vitesse de fonctionnement. La venue des mémoires caches, des mémoires virtuelles et des hiérarchies de mémoire a grandement augmenté la taille et la performance des mémoires. Pour cette raison, l'économie de mémoires ne constitue plus un facteur avantageux pour l'architecture CISC.

Autrefois, l'activité de programmation a été une tâche longue et fastidieuse pour les programmeurs. Cet aspect a longtemps encouragé la conception des processeurs CISC avec un code machine ressemblant le plus possible au langage de haut niveau. Tout en facilitant l'écriture des programmes en langage assembleur, les processeurs CISC exécutent une tâche avec un minimum d'instructions en obtenant ainsi de meilleures performances.

Depuis, l'évolution des logiciels a permis de réduire considérablement l'emploi des langages assembleurs au profit des langages hauts niveaux et d'accroître l'utilisation des compilateurs. Tout comme l'architecture des processeurs, l'optimisation des compilateurs est un facteur très influent sur la performance des processeurs. On constate que les compilateurs peuvent jouer un rôle tout aussi important que la fréquence d'horloge d'un processeur. Ces derniers ont permis aux concepteurs de mieux comprendre l'utilisation des processeurs et de développer une nouvelle vision architecturale basée sur des données empiriques, l'expérimentation et la simulation. Ils ont aussi éliminé le besoin de faciliter la programmation en langage assembleur.

Durant la dernière décennie, des études menées par le professeur D. A. Patterson de l'université de Berkeley (Hen, 1992) sur plusieurs architectures de processeurs ont démontré la sous-utilisation de plusieurs instructions. Celles qui sont les moins

utilisées sont généralement les plus complexes. Par exemple, dans le cas de l'ordinateur IBM 360, le tableau 1 extrait du livre de Hennessy et Patterson nous indique qu'en moyenne 88% du code exécuté par quatre programmes de test est réalisé par 32 instructions seulement. Pour le programme de test PLIGO, 13 instructions représentent 90% du code exécuté.

Ainsi, on peut penser qu'une réduction du jeu d'instructions optimise un processeur, c'est-à-dire qu'elle améliore la complexité, l'architecture et la performance. À cet effet, la loi d'Amdahl (Hen, 1992) nous indique qu'il y a une augmentation de la vitesse lorsqu'on améliore certaines parties d'un processeur. Le gain en vitesse obtenu ou l'accélération d'Amdahl est défini par :

$$accélération = \frac{temps\ sans\ utiliser\ l'amélioration}{temps\ avec\ l'amélioration\ lorsque\ possible} . \qquad (2)$$

D'après la loi d'Amdahl, pour un processeur, si on optimise son jeu d'instructions, on améliore sa performance et on réduit son jeu d'instructions. Pour exécuter une tâche, l'utilisation d'un processeur RISC nécessite l'usage d'un plus grand nombre d'instructions et de mémoire par programme. Pour les instructions les moins utilisées, on a certainement une dégradation de la performance. Toutefois, à cause de la grande occurrence des instructions rapides et fréquentes, on peut être sûr d'une réduction du temps d'exécution ($t$) et d'un gain de performance.

Les études de Patterson (Heu, 1990), en plus d'établir les instructions les plus utilisées, ont permis d'observer les cinq points suivants :

1- les instructions d'appels et de retours de procédures exigent le plus long temps d'exécution;
2- 80% des variables locales utilisées dans les programmes sont des scalaires;
3- 90% des structures complexes sont des variables globales;
4- la majorité des procédures n'échange que 6 arguments au maximum;
5- la profondeur maximale pour les appels de procédures est de 8 dans 99% des cas.

**TABLEAU 1**

**Les mesures détaillées de l'ordinateur 360**

| Instruction | PLIC | FORTGO | PLIGO | COBOLGO | Moyenne |
|---|---|---|---|---|---|
| **Contrôle** | **32 %** | **13%** | **5 %** | **16 %** | **16 %** |
| BC, BCR | 28 % | 13 % | 5 % | 14 % | 15 % |
| BAL, BALR | 3 % | | | 2 % | 1 % |
| **Arithmétique et logique** | **29 %** | **35 %** | **29 %** | **9 %** | **26 %** |
| A, AR | 3 % | 17 % | 21 % | | 10 % |
| SR | 3 % | 7 % | | | 3 % |
| SLL | | 6 % | 3 % | | 2 % |
| LA | 8 % | 1 % | 1 % | | 2 % |
| CLI | 7 % | | | | 2 % |
| NI | | | | 7 % | 2 % |
| C | 5 % | 4 % | 4 % | 0 % | 3 % |
| TM | 3 % | 1 % | | 3 % | 2 % |
| MH | | | 2 % | | 1 % |
| **Transferts de données** | **17 %** | **40 %** | **56 %** | **20 %** | **33 %** |
| L, LR | 7 % | 23 % | 28 % | 19 % | 19 % |
| MVI | 2 % | | 16 % | 1 % | 5 % |
| ST | 3 % | | 7 % | | 3 % |
| LD | | 7 % | 2 % | | 2 % |
| STD | | 7 % | 2 % | | 2 % |
| LPDR | | 3 % | | | 1 % |
| LH | 3 % | | | | 1 % |
| IC | 2 % | | | | 1 % |
| LTR | | 1 % | | | 0 % |
| **Virgule flottante** | | **7 %** | | | **2 %** |
| AD | | 3 % | | | 1 % |
| MDR | | 3 % | | | 1 % |
| **Décimaux, chaîne** | **4 %** | | | **40 %** | **11 %** |
| MVC | 4 % | | | 7 % | 3 % |
| AP | | | | 11 % | 3 % |
| ZAP | | | | 9 % | 2 % |
| CVD | | | | 5 % | 1 % |
| MP | | | | 3 % | 1 % |
| CLC | | | | 3 % | 1 % |
| CP | | | | 2 % | 1 % |
| ED | | | | 1 % | 0 % |
| **Totaux** | **82 %** | **95 %** | **90 %** | **85 %** | **88 %** |

Sur ces bases, une nouvelle approche de conception de processeurs appelée RISC a été fondée. On optimise l'architecture des processeurs RISC en utilisant des concepts très

modernes, tels que les fenêtres de registres, une multitude de registres, un jeu d'instructions réduit, une structure d'exécution *pipelinée* et l'optimisation de la rapidité des cas les plus fréquents. Les fenêtres de registres optimisent les appels et les retours de procédures. La multitude de registres permet la manipulation efficace des variables locales ou globales et des constantes. Le jeu d'instructions réduit favorise une architecture VLSI ( «*Very Large Scale Integrated circuits*») optimisée. La structure d'exécution *pipelinée* réduit le temps d'exécution ($t$). De plus, l'optimisation de la rapidité des opérations les plus fréquentes afin d'exécuter une instruction par cycle d'horloge augmente considérablement la performance des processeurs RISC.

Toutefois, il ne faut pas oublier l'existence d'autres approches architecturales extrêmement performantes, telle que l'architecture vectorielle et l'architecture parallèle. Un processeur vectoriel contient un processeur RISC auquel on a ajouté des instructions, des registres, une UAL (unité arithmétique et logique) et des unités de contrôle spécialisés permettant d'optimiser les opérations répétitives des langages évolués.

Dans l'architecture parallèle, on dispose de plusieurs processeurs qui travaillent en parallèle à l'exécution d'un programme. Dans certains cas, le processeur utilise un mot d'instructions très long qui lui permet d'exécuter plusieurs instructions en même temps, c'est-à-dire d'effectuer un traitement parallèle. Toutefois, ces deux approches nécessitent des circuits et des compilateurs très complexes.

L'architecture RISC a permis un progrès très remarquable dans le domaine informatique. Actuellement, des processeurs RISC commercialisés, tels qu'IBM-RS/6000, INTEL-I860, AMD-AM29000 et SUN-SPARC constituent la grande majorité des processeurs utilisés dans des stations de travail UNIX.

Ce mémoire a comme objectif principal la présentation d'un nouveau processeur RISC de 16 bits spécialisé pour la logique floue et d'un nouveau processeur flou. Elle vise aussi à présenter les différents aspects de sa conception : la conception du jeu d'instructions, l'organisation fonctionnelle, la conception logique et la matérialisation.

Telle que présentée à la figure 1, la méthodologie suivie pour la conception du processeur a été réalisée en 8 étapes :

Étape 1 - Conception du jeu d'instructions. Définition du jeu d'instructions et de son architecture pour un domaine d'application déterminé qui, dans ce cas, est la logique floue.

Étape 2 - Organisation fonctionnelle. Choix du système de mémorisation, du type de bus et du chemin de données. Définition de l'architecture interne du processeur.

Étape 3 - Description logique. Description des différentes unités qui composent notre conception avec le langage de description de matériels VHDL ( «*Very high speed integrated circuits Hardware Description Language*»).

Étape 4 - Simulation. Simulation du modèle VHDL avec les outils d'aide à la conception de Mentor Graphics sans tenir compte du délai. Le logiciel QUICKSIM a été utilisé. Retour à l'étape 3 afin de corriger notre modèle VHDL.

Étape 5 - Synthèse et Optimisation. Ces opérations sont réalisées avec le logiciel Autologic et la librairie des circuits FPGAs ( «*Field Programmable Gate Array*») de la famille XILINX-4000. Mise en oeuvre du processeur dans un circuit FPGA.

Étape 6 - Placement et Routage. Optimisation de notre conception dans l'architecture des circuits FPGAs reprogrammables de XILINX. Amélioration de la surface occupée et de la vitesse.

Étape 7 - Simulation du résultat avec QUICKSIM en tenant compte des délais engendrés par les caractéristiques matérielles des circuits FPGAs reprogrammables.

Étape 8 - Génération de fichiers permettant la configuration du circuit FPGA.

Le processeur présenté dans cet ouvrage utilise la plupart des concepts de l'architecture RISC et il est basé sur l'architecture des circuits FPGAs reprogrammables.

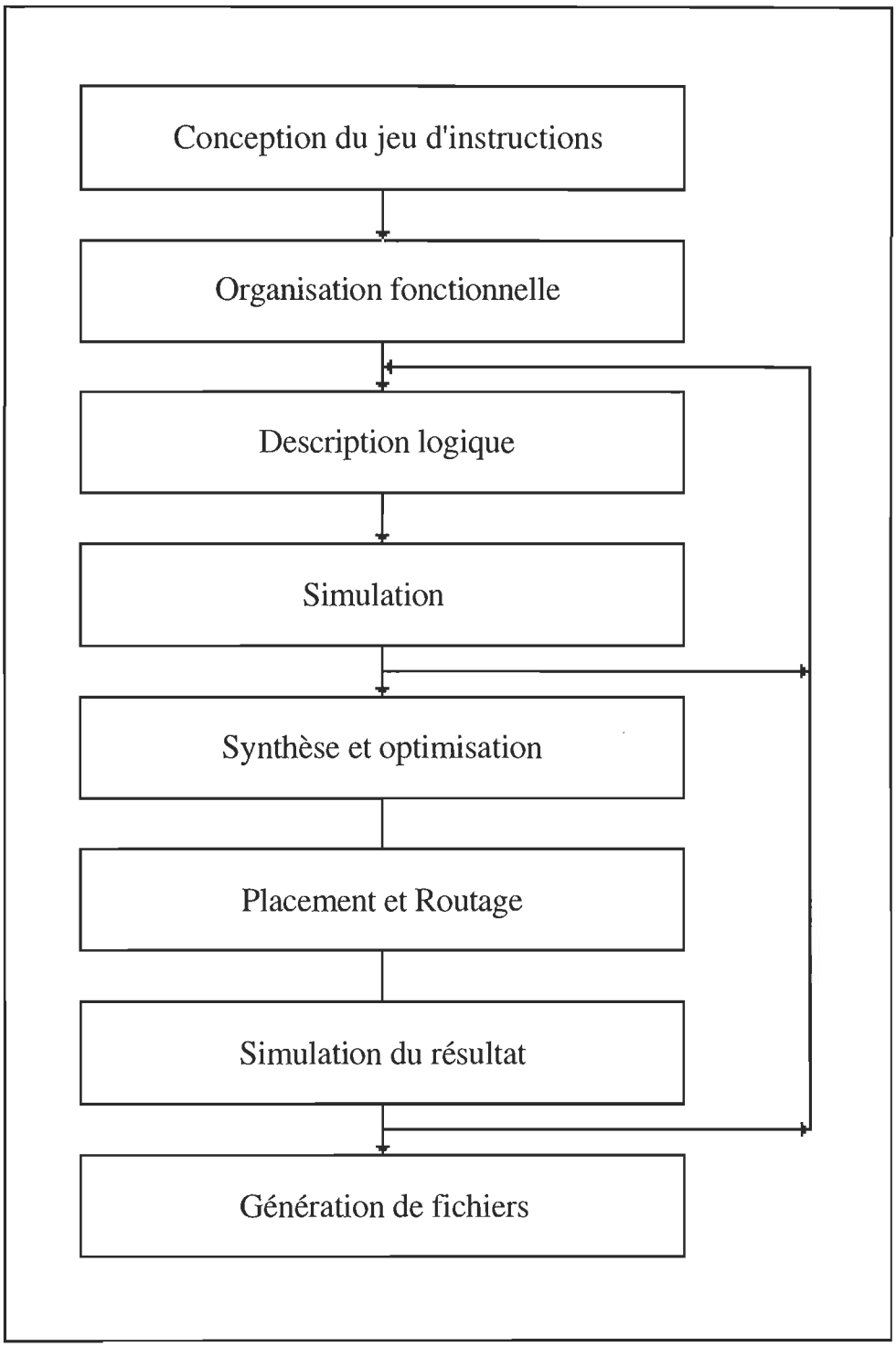


**Figure 1. Étapes de conception du processeur**

Le résultat de ce travail de recherche a été la réalisation de deux nouveaux processeurs : un processeur flou et un processeur RISC spécialisé pour la logique floue. Ce dernier a un jeu d'instructions qui lui permet d'optimiser les applications décrites avec la logique floue. Il a été mis en oeuvre avec des circuits FPGAs de la famille XILINX-4000 et il fonctionne avec une horloge de 10 Mhz. Le processeur RISC contient 81 registres et utilise 24 instructions et cinq modes d'adressage. Quant au processeur flou, il utilise la règle de composition «max-min», la méthode du centre de gravité (COG) et la technique de consultation de tableau.

Pour présenter cette conception, ce mémoire est divisé en cinq chapitres. Le premier chapitre présente l'architecture du jeu d'instructions et quelques notions de la logique floue. Le deuxième chapitre est dédié à l'organisation fonctionnelle du processeur. Le système de mémorisation, les structures de bus, le chemin de données et le traitement des interruptions y sont alors abordés. Le troisième chapitre expose la conception logique et la réalisation. Le quatrième chapitre détaille le processeur flou et, finalement, la conclusion est présentée au cinquième chapitre.

# CHAPITRE I

## CONCEPTION DU JEU D'INSTRUCTIONS

Le jeu d'instructions d'un processeur est une caractéristique très importante qui influence son architecture interne ainsi que sa performance. Pour classifier l'architecture d'un processeur, une méthode fiable consiste à utiliser l'UAL (Ham, 1985) qui, selon le type utilisé pour les accès de données à la mémoire, établit les trois catégories suivantes : l'architecture à pile, l'architecture à accumulateur et l'architecture à registres.

Une instruction permet de traiter une donnée située dans la mémoire ou dans un registre et cette donnée est appelée opérande. Dans l'architecture à pile, les opérandes sont empilés sur une pile qui est, en réalité, un registre d'indexation servant à sélectionner les cases mémoires contenant les valeurs des opérandes. Dans l'architecture à accumulateur, un des opérandes est l'accumulateur qui, en plus de pouvoir conserver une donnée, est capable d'effectuer des opérations arithmétiques et logiques telles que le décalage, l'incrémentation ou la négation. Dans l'architecture à registre, les opérandes sont des registres ou des cases mémoires rapides. Cette architecture favorise la manipulation efficace des variables. Elle permet aussi une meilleure flexibilité et une plus grande rapidité des calculs que les deux autres architectures.

Pour la conception du jeu d'instructions du processeur, on a choisi l'architecture à registres et on a décidé, conformément avec la méthodologie RISC, que les opérations arithmétiques et logiques s'effectueront sur des registres. Afin d'optimiser davantage notre conception, toutes les instructions ont un format fixe de 32 bits et les accès en mémoire sont effectués par des instructions spéciales.

Du point de vue matériel, une instruction est une série de bits regroupée ensemble dans

le registre d'instructions afin d'indiquer au processeur l'opération choisie par le programmeur. Généralement, ces bits sont divisés en plusieurs champs qui forment l'instruction. Tel que présenté à la figure 1.1, le format des instructions du processeur est composé de cinq champs indiquant les deux opérandes source et un opérande destination. Puisque ces trois opérandes sont des registres, le processeur utilise l'architecture à registres avec trois opérandes. Dans ce format, le premier champ, de cinq bits, contient le code d'opération, ce qui permet de spécifier jusqu'à 32 opérations. Le deuxième champ, d'un bit, spécifie le mode d'adressage immédiat si la valeur du bit est 1. Le troisième champ, de cinq bits, contient l'adresse du registre de destination ($R_d$) et permet donc de spécifier l'un des 32 registres visibles par le programmeur. Le quatrième champ, de cinq bits, contient l'adresse du premier registre de sources ($R_a$) et permet donc de spécifier l'un des 32 registres visibles par le programmeur. Ce quatrième champ peut aussi indiquer le mode d'adressage et sélectionner les registres ($F_0$ et $F_1$) servant à passer des paramètres ou contenir un des vecteurs d'interruptions logicielles (SWIVEC). Finalement, le cinquième champ, de 16 bits, contient l'adresse du deuxième registre de source ($R_b$) ou une valeur immédiate de 16 bits si le mode immédiat est utilisé. L'utilisation de ce format a permis d'obtenir un jeu d'instructions réduit à 24 instructions.

Il existe différentes méthodes pour spécifier l'adresse d'un opérande. Ces méthodes sont appelées les modes d'adressage. Le processeur utilise cinq modes d'adressage : le mode direct, le mode registre, le mode immédiat, le mode inhérent et le mode indirect. Dans le mode direct, l'adresse de l'opérande est indiquée directement dans l'instruction. Les bits 0 à 15 de l'instruction sont utilisés comme adresse. Dans le mode registre, toutes les opérations sont effectuées sur des registres ($R_a$, $R_b$, $R_d$) sans accéder à la mémoire. Les adresses de ces trois registres sont incluses dans l'instruction. Si on utilise l'opérande pour assigner une valeur à un registre alors on utilise le mode immédiat. Ce mode est surtout utilisé pour spécifier les constantes d'un programme. Lorsqu'une instruction n'a pas d'opérande, on utilise le mode inhérent. Finalement, le mode indirect est semblable au mode direct sauf que l'adresse de l'opérande est située dans le registre apparaissant dans l'instruction ($R_b$).

Dans l'architecture RISC, le jeu d'instructions doit être défini pour un domaine

d'application déterminé qui, dans notre cas, est la logique floue. À cet effet, on a introduit une catégorie d'instructions floues. Il est important de souligner qu'un concepteur d'architecture de jeu d'instructions doit effectuer des études quantitatives pour établir les instructions à fortes occurrences.

En se basant sur l'architecture de plusieurs processeurs, les instructions ont été regroupées en cinq catégories : les instructions arithmétiques et logiques, les instructions de branchements, les instructions d'accès à la mémoire, les instructions floues et les instructions d'interruptions logicielles. Toutes les instructions sont présentées au tableau 1.3.

## 1.1 Les instructions arithmétiques et logiques

Les instructions arithmétiques et logiques utilisent les modes d'adressage registre et immédiat. Ces instructions sont : ADD, SUB, XOR, ORR, AND, CMP, LSR, ROR et ROL. Elles permettent aussi l'émulation de plusieurs autres instructions. Par exemple, on émule l'instruction LSL (décalage à gauche) en additionnant un registre avec lui-même (ADD $R_d$, $R_b$, $R_b$). Ces opérations affectent l'état du processeur en modifiant les drapeaux N, C et Z (voir section 2.1.6).

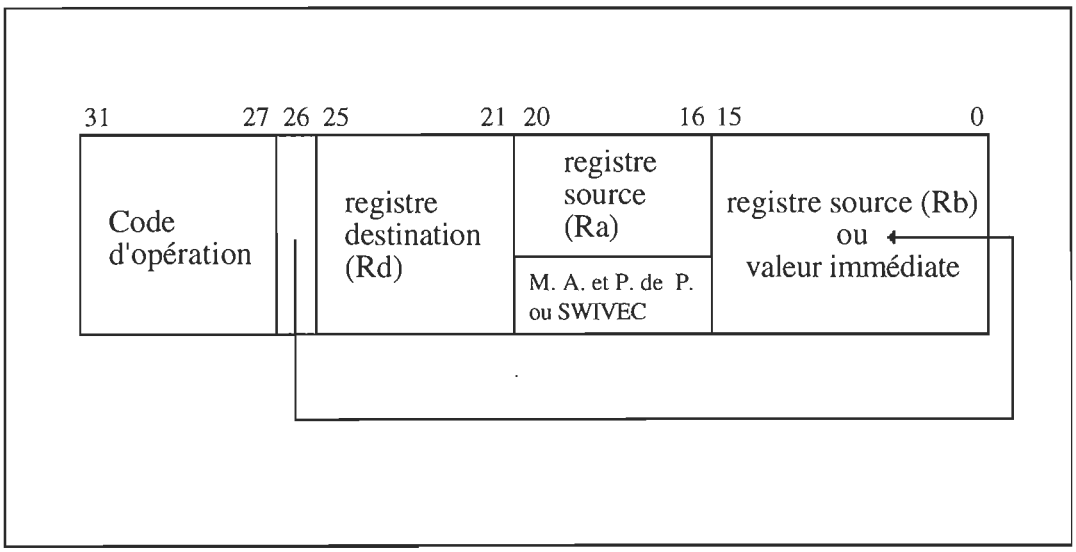


**Figure 1.1. Format des instructions du processeur**

## 1.2 Les instructions de branchements

Les instructions de branchements permettent d'effectuer un saut dans un programme. Il existe quatre sortes de branchements : les branchements inconditionnels, les branchements conditionnels, les appels de procédures et les retours de procédures.

### 1.2.1 Les branchements inconditionnels

Les branchements inconditionnels permettent un saut sans condition en modifiant la séquence du compteur de programme. Ils utilisent les modes d'adressage direct, registre, immédiat et indirect. Ils sont réalisés par l'instruction JMP.

### 1.2.2 Les branchements conditionnels

Le processeur utilise trois instructions de branchements conditionnels : BEQ, BMI et BCS. Le programme effectue un saut conditionné avec l'état du processeur et de l'instruction de branchements conditionnels choisie par le programmeur. BEQ est utilisé pour tester le drapeau Z, BMI, pour le drapeau N et BCS, pour le drapeau C. Par exemple, si le drapeau C est à 1 et le processeur exécute l'instruction BCS alors la séquence du compteur de programme est modifiée donc un saut conditionnel est effectué. Les instructions de branchement conditionnel permettent de réaliser les boucles et les instructions conditionnelles dans les langages évolués ou de faire un choix parmi plusieurs possibilités.

### 1.2.3 Les appels de procédures

L'instruction JSR effectue les appels de procédures. Ces derniers facilitent la compréhension d'un programme en créant une meilleure structure. L'instruction JSR facilite aussi les opérations de passages de paramètres ainsi que les opérations de sauvegarde de registres. Cette instruction utilise les modes d'adressage direct, immédiat, registre et indirect.

### 1.2.4 Les retours de procédures

Les retours de procédures, réalisés par l'instruction RTS, permettent le retour d'une procédure à un sous-programme tout en facilitant les passages de paramètres et les opérations de restauration de registres. Cette instruction utilise le mode d'adressage inhérent.

## 1.3 Les instructions d'accès à la mémoire

Il arrive souvent qu'on copie des opérandes et des résultats entre la mémoire et un registre du processeur. Cette copie nécessite deux types d'accès à la mémoire : l'accès en écriture et l'accès en lecture. L'instruction STR effectue la copie du contenu d'un registre vers la mémoire et elle réalise ainsi l'accès en écriture. Elle utilise les modes d'adressage direct et indirect. L'accès en lecture est effectué par l'instruction LDR qui permet la copie du contenu d'une case mémoire vers un registre. Cette instruction utilise les modes d'adressage direct, registre, immédiat et indirect.

## 1.4 Les instructions floues

N'étant pas nécessairement vraies ou fausses, les informations vagues, imprécises ou approximatives sont mieux décrites avec la logique floue qu'avec la logique conventionnelle. La logique floue autorise un changement graduel ou des états intermédiaires dans le passage d'un état à un autre. Elle permet aussi une appartenance partielle d'un élément dans une classe donnée et elle a une infinité de niveaux compris entre 0 et 1. La logique floue représente une généralisation de la logique booléenne. Toutefois, les systèmes flous utilisent une approche très différente des systèmes conventionnels. Par exemple, en automatisme, un système ordinaire exécute un algorithme mathématique qui lui permet de déterminer l'action à prendre en fonction de l'état d'un processus, tandis qu'un système flou vise plutôt la modélisation du comportement d'un opérateur humain pouvant gérer ce processus. Cette approche permet de simuler le comportement humain et elle est simple et très efficace.

Considérons un sous-ensemble $S$ de $U$, tel que $S \subset U$. Tout élément $e$ de $U$ peut

appartenir ou ne pas appartenir à $S$. Donc, on peut associer une fonction caractéristique $f_S$, tel que :

$$f_S: U \rightarrow \{0,1\}, \tag{1.1}$$

c'est-à-dire que pour tout élément $e$ de $U$ :

$$\left.\begin{array}{l} f_S(e) = 1 \; si \; e \in S \\ f_S(e) = 0 \; si \; e \notin S \end{array}\right\} \tag{1.2}$$

Ce sous-ensemble est dit vulgaire, classique ou ordinaire. Maintenant, pour introduire la notion d'appartenance partielle, on définit un sous-ensemble $T$ de $U: T \subset U$ à laquelle on associe une fonction caractéristique $\mu_t$, tel que :

$$\mu_t: U \rightarrow [0,1], \tag{1.3}$$

c'est-à-dire que pour tout élément $e$ de $U$ :

$$\left.\begin{array}{l} \mu_t(e) = 0, \textit{ si e n' a aucune appartenence à T}, \\ \mu_t(e) = 1, \textit{ si e a une appartenence totale à T}, \\ 0 < \mu_t(e) < 1, \textit{ si e a une appartenence partielle à T}. \end{array}\right\} \tag{1.4}$$

Le sous-ensemble $T$ est un sous-ensemble flou et la valeur de la fonction caractéristique $\mu_t(e)$ s'appelle le degré d'appartenance de $e$ à $T$. La théorie des sous-ensembles flous permet de représenter et de manipuler des concepts ou des données vagues, incertains ou subjectifs. Par exemple, les concepts chaud et tiède sont représentés à la figure 1.2. Ils sont appelés les étiquettes ou les variables linguistiques de ce sous-ensemble. Dans cette même figure, on peut remarquer deux fonctions d'appartenance et leur degré d'appartenance ( $\mu_t(70)=(0.8, 0)$) dans leur domaine d'application qui, dans ce cas, est la température. Une valeur exacte ($70^\circ$ F) est aussi représentée.

Deux sous-ensembles flous $X$ et $Y$ de $E$ sont dits égaux si et seulement si pour tout

élément $e$ dans $E$, les degrés d'appartenance sont égaux, c'est-à-dire que :

$$\forall e \in E: \mu_X(e) = \mu_Y(e), \tag{1.5}$$

et on le note par $X = Y$. Sinon, $X$ est différent de $Y$ et ceci est noté par $X \neq Y$.

$X$ est le complément de $Y$, si et seulement si pour tout élément $e$ dans $E$, la somme des degrés d'appartenance est égale à 1, c'est-à-dire que :

$$\forall e \in E: \mu_X(e) = 1 - \mu_Y(e), \tag{1.6}$$

et on le note par $X = \overline{Y}$.

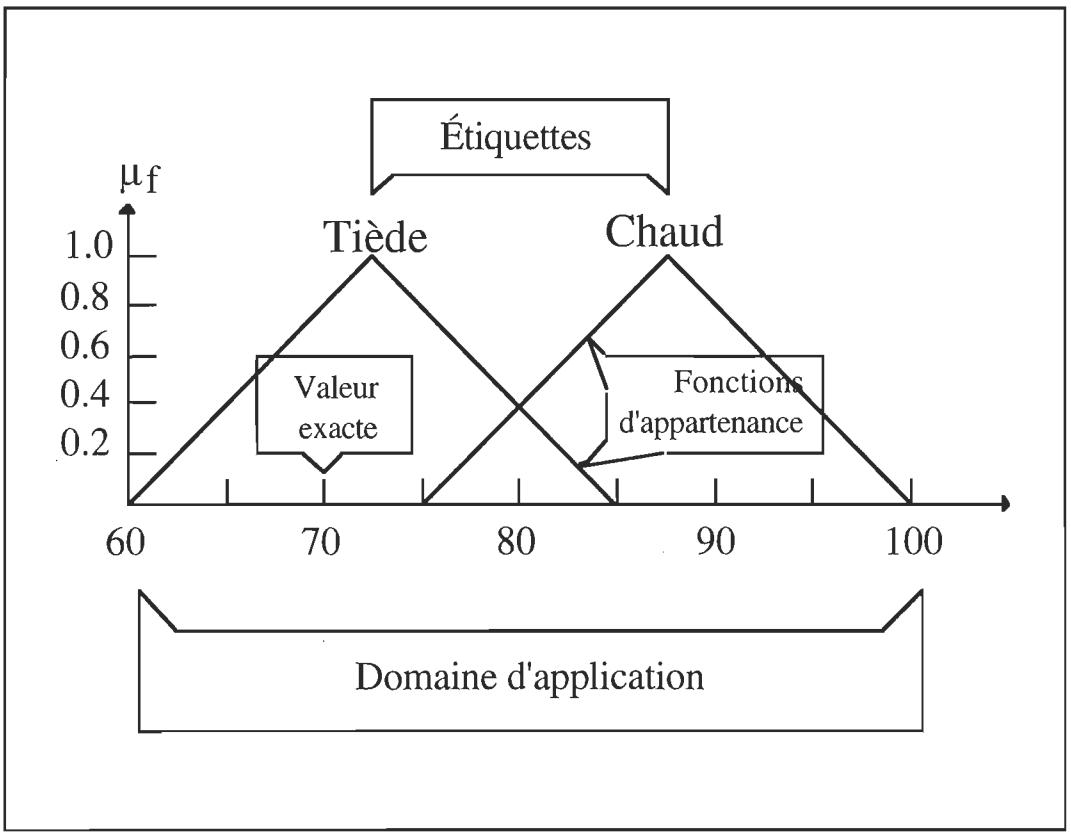


**Figure 1.2. Sous-ensembles flous**

$X$ est inclus dans $Y$, si et seulement si pour tout élément $e$ dans $E$, on a que le degré d'appartenance de $X$ est inférieur ou égal à celui de $Y$, c'est-à-dire que :

$$\forall e \in E: \mu_X(e) \leq \mu_Y(e), \tag{1.7}$$

et on le note par $X \subseteq Y$. Sinon X n'est pas inclus dans $Y$ et ceci est noté par $X \not\subset Y$.

On définit l'intersection de $X$ et $Y$ par un sous-ensemble $Z$ contenant le degré d'appartenance minimal entre $X$ et $Y$ pour tous les éléments $e$ dans $E$, c'est-à-dire que :

$$\forall e \in E: \mu_Z = \min\left(\mu_X(e), \mu_Y(e)\right), \tag{1.8}$$

et on le note par $Z = X \cap Y$. À la figure 1.3, la partie ombrée représente l'intersection de $X$ et $Y$.

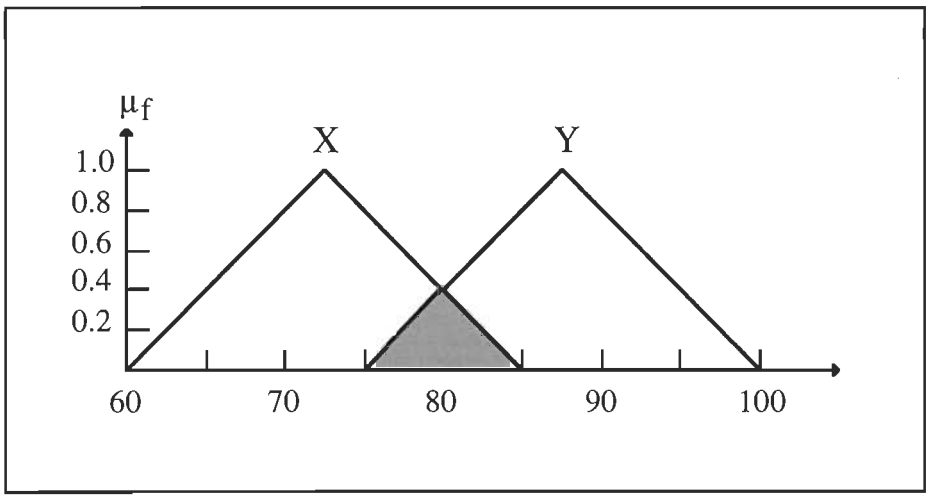


**Figure 1.3. Intersection de deux sous-ensembles flous**

On définit l'union de $X$ et $Y$ par un sous-ensemble $Z$ contenant le degré d'appartenance maximal entre $X$ et $Y$ pour tout élément $e$ dans $E$, c'est-à-dire que :

$$\forall e \in E: \mu_Z(e) = \max\left(\mu_X(e), \mu_Y(e)\right), \tag{1.9}$$

et on le note par $Z = X \cup Y$ . L'union de deux sous-ensembles flous est représentée à la figure 1.4.

On peut utiliser des opérateurs autres que le maximum et le minimum pour définir l'union et l'intersection de deux sous-ensembles flous. Dans le tableau 1.1, on a regroupé quelques-uns de ces opérateurs.

**TABLEAU 1.1**
**La liste de quelques opérateurs**

| **Intersection** | **Nom de l'opérateur d'intersection** | **Union** | **Nom de l'opérateur d'union** |
|---|---|---|---|
| $\min(x,y)$ | Zadeh | $\max(x,y)$ | Zadeh |
| $x \bullet y$ | Produit algébrique | $x + y - x \bullet y$ | Somme algébrique |
| $\min((x+y),1)$ | Produit borné | $\max((x+y-1),0)$ | Somme bornée |
| $\begin{cases} x & si\ y = 1, \\ y & si\ x = 1, \\ 0 & autrement \end{cases}$ | Produit drastique | $\begin{cases} x & si\ y = 0, \\ y & si\ x = 0, \\ 1 & autrement \end{cases}$ | Somme drastique |

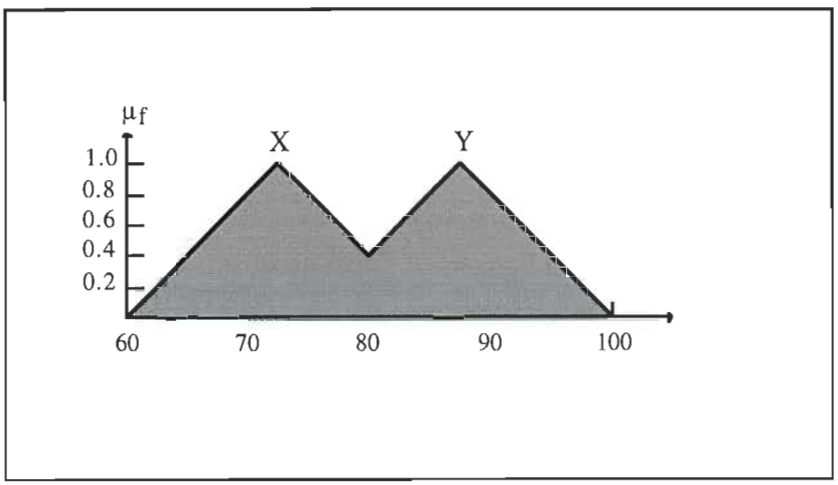


**Figure 1.4. Union de deux sous-ensembles flous**

Les principales propriétés des sous-ensembles flous sont les suivantes :

$$\left\{\begin{matrix} X \cap Y = Y \cap X \\ X \cup Y = Y \cup X \end{matrix}\right\} \quad \textit{Commutativité}, \tag{1.10}$$

$$\left\{\begin{matrix} (X \cap Y) \cap Z = X \cap (Y \cap Z) \\ (X \cup Y) \cup Z = X \cup (Y \cup Z) \end{matrix}\right\} \quad \textit{Associativité}, \tag{1.11}$$

$$\left\{\begin{matrix} X \cap X = X \\ X \cup X = X \end{matrix}\right\} \quad \textit{Idempotence}, \tag{1.12}$$

$$\left\{\begin{matrix} X \cap (Y \cup Z) = (X \cap Y) \cup (X \cap Z) \\ X \cup (Y \cap Z) = (X \cup Y) \cap (X \cup Z) \end{matrix}\right\} \quad \textit{Distributivité}, \tag{1.13}$$

$$X \cap \varnothing = \varnothing, \tag{1.14}$$

$$X \cup \varnothing = X, \tag{1.15}$$

$$X \cap E = X, \tag{1.16}$$

$$X \cup E = E, \tag{1.17}$$

$$\left\{\begin{matrix} \overline{(\overline{X})} = X \\ \overline{X \cap Y} = \overline{X} \cup \overline{Y} \\ \overline{X \cup Y} = \overline{X} \cap \overline{Y} \end{matrix}\right\} \quad \textit{Loi de Morgan.} \tag{1.18}$$

On appelle une relation de $X$ vers $Y$, une correspondance $(R)$ qui associe un ou plusieurs éléments de $Y$ à certains éléments de $X$. Cette relation est dite floue si et seulement si le sous-ensemble $R$ est flou. La relation s'écrira alors :

$$x \in X,\ y \in Y:\ x \mathfrak{R} y. \tag{1.19}$$

La composition «max-min» de relations floues $\mathfrak{R}_1 \subset X \times Y$ et $\mathfrak{R}_2 \subset Y \times Z$, notée par $\mathfrak{R}_1 \circ \mathfrak{R}_2$, est définie par :

$$\mu_{\mathfrak{R}_1 \circ \mathfrak{R}_2}(x,z) = \max_y \left[ \min\left( \mu_{\mathfrak{R}_1}(x,y), \mu_{\mathfrak{R}_2}(y,z) \right) \right]. \tag{1.20}$$

On peut substituer l'opérateur «min» par un autre opérateur afin d'obtenir d'autres types de composition. Par exemple, en le remplaçant par le produit algébrique, on

obtient la composition «max-produit». Nous nous restreignons à la composition «max-min» vu qu'elle est la plus simple à implanter et la plus utilisée. Lorsque $X$ et $Y$ sont finis, $X \times Y$ peut être vu comme un produit matriciel où on remplace l'addition par l'opérateur «max» et la multiplication, par l'opérateur «min». La composition «max-min» est illustrée à la figure 1.5 et elle permet, pour les applications décrites avec la logique floue, de déterminer la sortie floue. Si $x \in X$, $y \in Y$ et $R$ est une relation floue de $X$ vers $Y$, on peut déduire $y$ de $Y$ en utilisant la composition de $R$ avec $x$ (voir fig. 1.5). C'est ce qu'on appelle l'implication floue et on l'obtient par :

$$y = x \circ R. \tag{1.21}$$

L'implication floue équivaut à une règle floue définie par :

$$Si\ x\ Alors\ y. \tag{1.22}$$

Les instructions floues visent à optimiser notre processeur pour les informations décrites avec la logique floue. Un processeur RISC spécialisé pour la logique floue, et, plus précisément pour celle de Zadeh, contient un jeu d'instructions réduit auquel on a ajouté les instructions MAX et MIN.

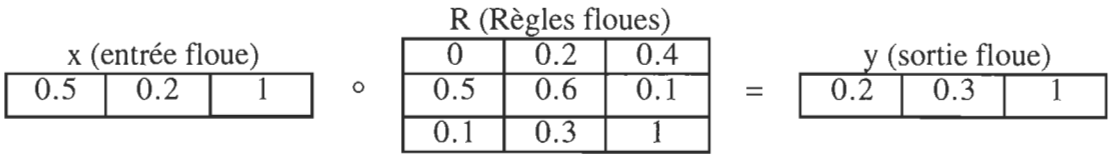


**Figure 1.5 Composition max-min**

Dans le tableau 1.2, lorsque $R_1$ est inférieur à $R_2$, MIN remplace trois instructions, dans le cas contraire, elle remplace 4 instructions. Ceci est aussi applicable à l'instruction MAX. Donc, on peut dire que les instructions floues remplacent en moyenne 3.5 instructions. Selon les résultats expérimentaux de Watanabe et D. Chen (Wat, 1993), l'implantation de ces instructions dans un processeur RISC conventionnel peut augmenter la vitesse jusqu'à un facteur de 2.5. Ces instructions sont exécutées par l'UAL mais leurs résultats ne modifient pas l'état du processeur.

**TABLEAU 1.2**
**Émulation de l'instruction MIN**

| Étiquettes | Instructions | Opérandes |
|---|---|---|
| | CMP | $R_1, R_2$ |
| | BMI | PETIT |
| | LDR | $R_2, R_3$ |
| | JMP | SUITE |
| PETIT | LDR | $R_1, R_3$ |
| SUITE | <prochaine instruction> | |

## 1.5 Les instructions d'interruptions logicielles

Les instructions d'interruptions logicielles permettent une gestion efficace des interruptions. Ces instructions sont : SEI, CLI, SWI et RTI. Les instructions SEI et CLI permettent de mettre le drapeau I à 1 ou à 0 respectivement. SWI sauvegarde l'adresse de retour, met le drapeau I à 1 et effectue un branchement à un des vecteurs d'interruptions SWIVEC. Ces vecteurs sont au nombre de quatre et ils sont situés aux adresses 8, 10, 12 et 14.

Similairement à l'instruction SWI, les interruptions matérielles sauvegardent l'adresse de retour, forcent le drapeau I à 1 et effectuent un branchement au vecteur NMIVEC (adresse 4) pour NMI ou IRQVEC (adresse 2) pour IRQ. Toutefois, contrairement à l'interruption NMI qui est toujours autorisée, l'interruption IRQ peut intervenir seulement quand le processeur n'est pas déjà en interruption et lorsqu'elle est autorisée par le programmeur c'est-à-dire que le drapeau IRQ à 1. Finalement, RTI effectue le retour d'une sous-routine d'interruptions, rétablit l'adresse de retour et fait une remise à 0 du drapeau I. Toutes ces instructions utilisent le mode d'adressage inhérent.

# TABLEAU 1.3

## Les instructions du processeur

| Code Op. | Instruction | Format | UAL | Description |
|---|---|---|---|---|
| 00 | No operation | NOP | PB | |
| 01 | Addition | ADD $R_d$, $R_a$, [#]$R_b$ | AL | $R_d \leftarrow R_a$ + [#]$R_b$ |
| 02 | Soustraction | SUB $R_d$, $R_a$, [#]$R_b$ | AL | $R_d \leftarrow R_a$ - [#]$R_b$ |
| 03 | Comparaison | CMP $R_a$, [#]$R_b$ | AL | $R_a$ - [#]$R_b$ |
| 04 | Ou Exclusif | XOR $R_d$, $R_a$, [#]$R_b$ | AL | $R_d \leftarrow R_a \oplus$ [#]$R_b$ |
| 05 | Ou logique | ORR $R_d$, $R_a$, [#]$R_b$ | AL | $R_d \leftarrow R_a \vee$[ #]$R_b$ |
| 06 | Et logique | AND $R_d$, $R_a$, [#]$R_b$ | AL | $R_d \leftarrow R_a \wedge$ [#]$R_b$ |
| 07 | Shift Right | LSR $R_d$, $R_a$, [#]$R_b$ | AL | $R_d \leftarrow R_a >>$ [#]$R_b$ |
| 08 | Rotate Right | ROR $R_d$, [#]$R_b$ | AL | $R_d \leftarrow$ ROR( [#]$R_b$ ) |
| 09 | Rotate Left | ROL $R_d$, [#]$R_b$ | AL | Rd $\leftarrow$ ROL( [#]$R_b$ ) |
| 0A | Minimum | MIN $R_d$, $R_a$, [#]$R_b$ | AL | $R_d \leftarrow$MIN($R_a$,[#]$R_b$ ) |
| 0B | Maximum | MAX $R_d$, $R_a$, [#]$R_b$ | AL | $R_d \leftarrow$MAX($R_a$,[#]$R_b$ ) |
| 10 | Jump | JMP [#]$R_b$ | PB | PC $\leftarrow$[#]$R_b$ |
| | | JMP ,$R_b$ | PD | PC $\leftarrow$M[$R_b$] |
| | | JMP adresse | PD | PC $\leftarrow$M[adresse] |
| 11 | Jump If Egal | BEQ [#]$R_b$ | PB | Si Z=1 Alors PC $\leftarrow$[#]$R_b$ |
| | | BEQ ,$R_b$ | PD | Si Z=1 Alors PC $\leftarrow$M[$R_b$] |
| | | BEQ adresse | PD | Si Z=1 Alors PC $\leftarrow$M[adresse] |
| 12 | Jump If Minus | BMI [#]$R_b$ | PB | Si N=1 Alors PC $\leftarrow$[#]$R_b$ |
| | | BMI ,$R_b$ | PD | Si N=1 Alors PC $\leftarrow$M[$R_b$] |
| | | BMI adresse | PD | Si N=1 Alors PC $\leftarrow$M[adresse] |
| 13 | Jump if Carry | BCS [#]$R_b$ | PB | Si C=1 Alors PC $\leftarrow$[#]$R_b$ |
| | | BCS ,$R_b$ | PD | Si C=1 Alors PC $\leftarrow$M[$R_b$] |
| | | BCS adresse | PD | Si C=1 Alors PC $\leftarrow$M[adresse] |
| 14 | Jump of SUB | JSRP [#]$R_b$ | PB | PTR $\leftarrow$PTR+1, PC $\leftarrow$[#]$R_b$ , pas. de par. |
| | | JSRP ,$R_b$ | PD | PTR $\leftarrow$PTR+1, PC $\leftarrow$M[$R_b$], pas. de par. |
| | | JSRP adresse | PD | PTR $\leftarrow$PTR+1, PC $\leftarrow$M[adresse], pas. de pa. |
| 15 | Software INT. | $SWI_N$ | PB | PTR $\leftarrow$PTR+1, PC $\leftarrow$SWIVEC $\bullet$ 2, I $\leftarrow$1 |
| 16 | Ret. From SUB | $RTS_P$ | PB | PTR $\leftarrow$PTR-1, passage de paramètres |
| 17 | Ret. From INT. | RTI | PB | PTR $\leftarrow$PTR-1, I $\leftarrow$0 |
| 18 | Load | LDR $R_d$,[#]$R_b$ | PB | $R_d \leftarrow$[#]$R_b$ |
| | | LDR $R_d$,[$R_b$] | PD | $R_d \leftarrow$M[$R_b$] |
| | | LDR $R_d$,adresse | PD | $R_d \leftarrow$M[adresse] |
| 19 | Store | STR $R_b$, adresse | PB | M[adresse] $\leftarrow R_b$ |
| | | STR $R_b$, [$R_a$] | PB | M[$R_a$] $\leftarrow R_b$ |
| 1A | Set I | SEI | PB | I $\leftarrow$1 |
| 1B | Clear I | CLI | PB | I $\leftarrow$0 |
| --- | Interruption | NMI | PB | Si MNI=1 Alors PTR $\leftarrow$PTR+1, PC $\leftarrow$NMIVEC, I $\leftarrow$1 |
| | Hardware | IRQ | PB | Si I=0 et IRQ=1 Alors PTR $\leftarrow$PTR+1, PC $\leftarrow$IRQVEC, I $\leftarrow$1 |

# CHAPITRE II

# ARCHITECTURE

Un aspect important dans la conception d'un processeur est l'organisation fonctionnelle. Elle permet d'établir le système de mémorisation, le type de bus et le chemin de donnée. Elle définit ainsi l'architecture interne du processeur (Hen, 1994).

## 2.1 Le système de mémorisation

La mémoire centrale est constituée d'un ensemble de cases mémoires contenant chacune des informations binaires. Dans notre étude, on a utilisé une mémoire ayant une capacité de 64 Kmots (65 536 cases mémoires) où chaque case mémoire contient un mot de 16 bits. Pour accéder à un mot, on doit connaître son emplacement c'est-à-dire son adresse. Avec un pointeur d'adresses de 16 bits, on peut sélectionner n'importe quel mot de la mémoire centrale externe au processeur.

La copie de données entre le processeur et la mémoire peut s'effectuer en mode synchrone ou en mode asynchrone. Dans le mode asynchrone, le processeur fait une demande d'accès et il attend la réponse de la mémoire. En lecture, la mémoire envoie un signal indiquant au processeur que la donnée est valide et qu'elle peut être lue. En écriture, elle envoie un signal signifiant l'acquisition de la donnée. Dans le mode synchrone, le processeur envoie, dans un premier temps, un signal de lecture ou d'écriture à la mémoire. Ensuite, deux horloges effectuent le transfert : une est utilisée pour l'accès en lecture et l'autre, pour l'accès en écriture. Le mode synchrone, contrairement au mode asynchrone, ne permet pas le fonctionnement de deux composants à des vitesses différentes, sauf si on les synchronise sur la vitesse du plus lent. Puisque les mémoires actuelles sont très rapides, on a retenu pour la conception du processeur le transfert synchrone à cause de sa simplicité.

Pour compléter l'étude du système de mémorisation, il est important d'aborder le système de mémorisation temporaire de données qui est formé de 81 registres : le registre d'instructions IR, le compteur de programme PC, les 21 registres de données ($D_0$ à $D_{20}$), le pointeur de la pile SP, le pointeur de fenêtres PTF, le registre de statut et de contrôle SCR et la roue de fenêtres formée de 56 registres (voir fig. 2.4). Tous ces registres sont de 16 bits sauf SCR (5 bits) et IR (32 bits) et ils sont représentés à la figure 2.1. Puisque les registres sont des cases mémoires rapides, on peut alors accéder à leur contenu en spécifiant leur adresse ou un nom équivalent à celle-ci. Par exemple, on spécifie le registre situé à l'adresse 01 par son nom $D_0$ ou encore par son adresse $R_1$. On utilise un bus d'adresse de cinq bits pour accéder aux différents registres. Ainsi, seulement 32 des 81 registres sont accessibles par une instruction donnée.

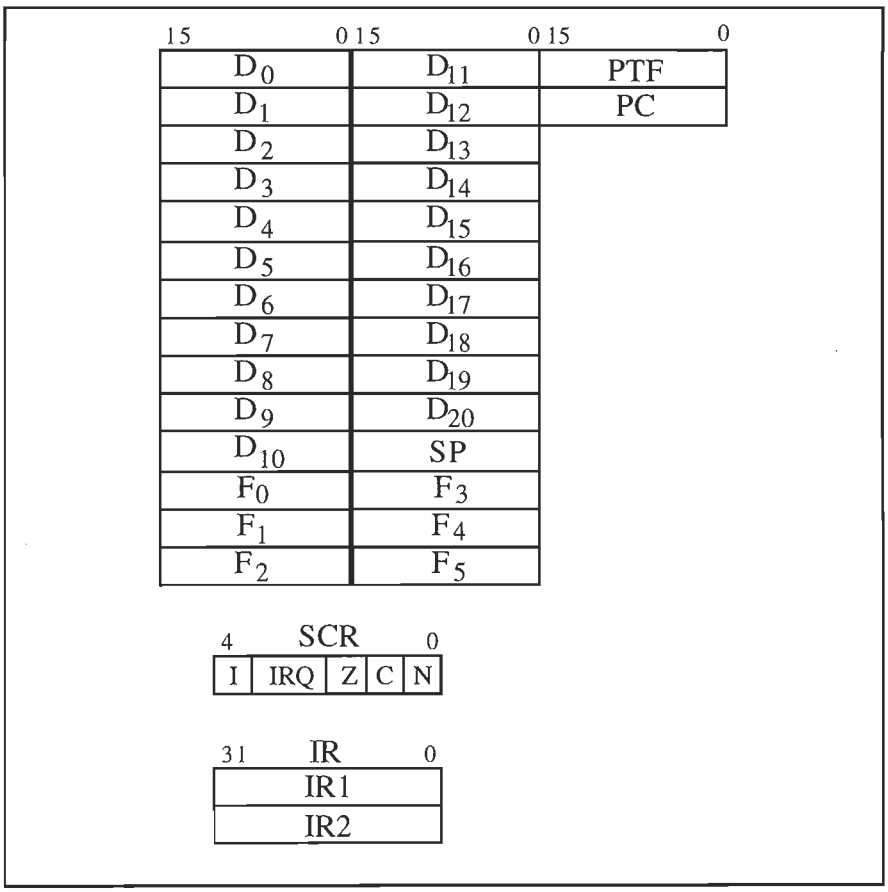


**Figure 2.1. Registres de programmation du processeur**

### 2.1.1 Le registre d'instructions IR

Le registre d'instructions IR est situé à l'adresse 0 et il est constitué de deux registres de 32 bits : IR1 et IR2. Ce dernier contient l'instruction courante et IR1 contient la prochaine instruction à exécuter. La sortie du registre IR est reliée aux circuits de contrôle afin de générer les signaux de séquencements (chronogrammes) qui permettent l'exécution d'une instruction. Ce registre permet aussi d'obtenir une structure d'exécution *pipelinée* à deux étages. IR est schématisé à la figure 2.2 et le pipeline, à la figure 2.3.

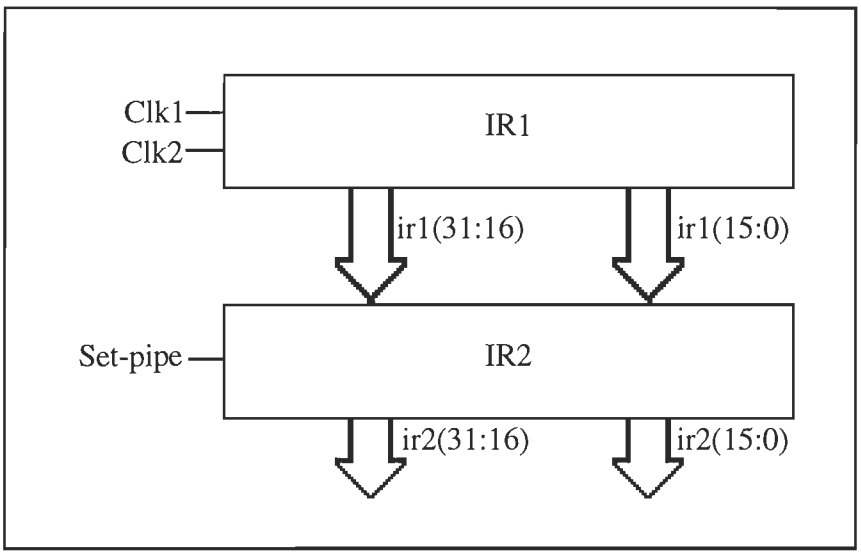


**Figure 2.2. Registre d'instructions IR**

Un pipeline est une technique qui permet l'exécution dans le même temps de plusieurs instructions. Puisque les instructions ont une structure d'exécution séquentielle, le processeur exécute chaque instruction en deux parties indépendantes. Premièrement, il effectue la lecture et le décodage de l'instruction (FETCH/DÉCODE) et il l'exécute ensuite (EXÉCUTE). Il peut, en chevauchant les instructions, exécuter deux séquences en même temps. Cette méthode a ainsi permis d'obtenir une structure d'exécution *pipelinée* à deux étages.

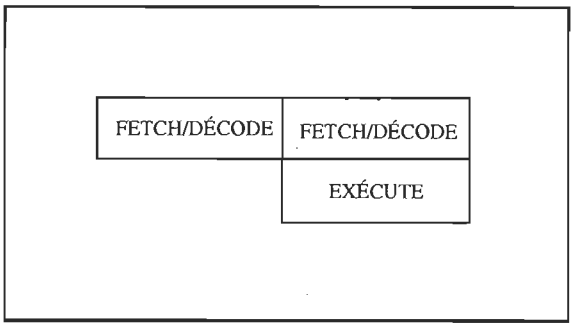


**Figure 2.3. Structure d'exécution pipelinée à deux étages**

Cependant, il peut arriver des situations qui brisent cette structure. C'est ce qu'on appelle des aléas. Il existe trois sortes d'aléas : les aléas structurels, les aléas de données et les aléas de contrôle. Les aléas structurels sont causés par des conflits de ressources quand le processeur ne peut pas gérer toutes les combinaisons possibles de chevauchements d'instructions. Lorsqu'une instruction attend le résultat de l'instruction précédente, il se produit un aléa de données. Les aléas de contrôle sont provoqués par les instructions qui modifient la séquence d'adressage des instructions subséquentes.

### 2.1.2 Le compteur de programme PC

Le compteur de programme PC ($R_{30}$) est un registre qui pointe les instructions à exécuter dans la mémoire. Il est mis à jour tout au long du processus d'exécution.

### 2.1.3 Les registres de données ($D_0$ à $D_{20}$)

Le processeur dispose de 21 registres de données à usage général : $D_0$ à $D_{20}$ ($R_1$ à $R_{21}$). Ils permettent d'effectuer un grand nombre d'opérations internes c'est-à-dire sans accéder à la mémoire centrale et d'obtenir une manipulation efficace des variables globales ainsi que des constantes.

### 2.1.4 La roue de fenêtres

La roue de fenêtres est formée de 8 fenêtres contenant chacune six registres $F_0$ à $F_5$

($R_{24}$ à $R_{29}$) et un PC ($R_{30}$). Les appels et les retours de procédures des langages évolués exigent un long temps d'exécution car, lors de l'exécution de ces instructions, le processeur doit conserver ou rétablir l'adresse de retour, passer des paramètres et effectuer le branchement à la procédure voulue. La roue permet de réaliser toutes ces opérations en un seul cycle d'horloge en réduisant ainsi le coût supplémentaire des appels et des retours de procédures.

La fenêtre courante est celle pointée par le pointeur de fenêtre. Lors des appels, la roue se déplace afin de présenter une nouvelle fenêtre courante. Elle sauvegarde ainsi le contexte de la procédure appelante et elle effectue ensuite le branchement à la procédure voulue.

Pour un retour, la roue se déplace dans l'autre direction et elle peut alors rétablir le contexte de la procédure appelante. En même temps, la roue permet d'effectuer le passage de deux paramètres au maximum.

Une seule fenêtre est accessible dans un temps donné c'est-à-dire que seulement 7 des 56 registres sont utilisés par une instruction. Pour les imbrications supérieures à 8, elle génère une exception permettant de sauvegarder ou de rétablir le contenu des registres de la fenêtre courante, ce qui élimine toute limitation à la profondeur des appels. Les registres $F_0$ à $F_5$ sont aussi utilisés pour contenir des variables locales et pour passer des paramètres ($F_0$ et $F_1$). La roue est représentée à la figure 2.4.

### 2.1.5 Le pointeur de pile SP et le pointeur de fenêtres PTF

Le pointeur de fenêtres PTF ($R_{23}$) contient l'adresse de la fenêtre courante et le pointeur de pile SP ($R_{22}$) pointe à l'adresse de la mémoire où le contenu des registres est sauvegardé. Ces registres peuvent être aussi incrémentés ou décrémentés. Seulement les bits 0 à 2 de PTF sont utilisés pour pointer la fenêtre courante, ainsi, en passant de 0 à 7 ou de 7 à 0 et en incrémentant ou en décrémentant, il permet de simuler un fonctionnement sous forme de roue des fenêtres. Les autres bits de ce registre servent à indiquer un débordement de la capacité réelle de la roue. Ce débordement intervient lorsqu'on le nombre d'appels sans retours dépassent 8.

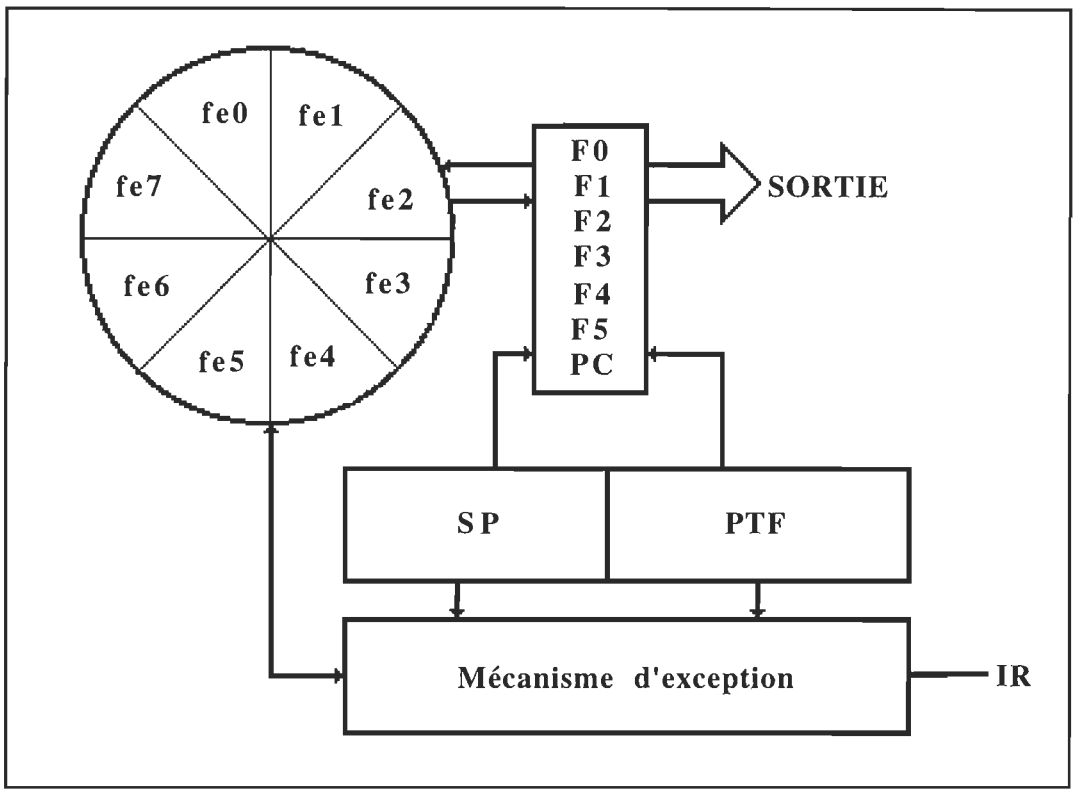


**Figure 2.4. La roue de fenêtres**

### 2.1.6 Le registre de statut et de contrôle SCR

Le registre de statut et de contrôle SCR ($R_{31}$) contient les différents drapeaux indiquant les états du système : N, C, Z, I et IRQ. Les drapeaux N, C et Z fournissent des informations sur les résultats de l'UAL. Le drapeau N indique un résultat négatif et le drapeau Z indique un résultat nul. Quant au drapeau C, il indique la retenue arithmétique. Lorsque à 1, le drapeau IRQ autorise l'interruption matérielle IRQ et le drapeau I indique la présence d'une interruption matérielle ou logicielle.

## 2.2 La structure de bus

Un processeur doit être capable de communiquer avec divers périphériques en plus de la mémoire centrale. Cela nécessite un certain nombre de connexions servant à établir

des liens de communications qui sont appelés «bus». Il existe trois types de «bus» : les bus de données, les bus d'adresse et les bus de contrôle. Un bus de données transmet un mot d'informations binaires. Un bus d'adresse indique l'emplacement d'un mot dans la mémoire centrale et finalement un bus de contrôle établit les différents contrôles de séquencements telles que la direction de la copie d'une donnée et la validité des adresses.

Un processeur peut utiliser une structure à deux bus. Telle que présentée à la figure 2.5, cette structure permet au processeur d'utiliser un premier bus pour communiquer avec la mémoire et un deuxième bus pour communiquer avec les périphériques.

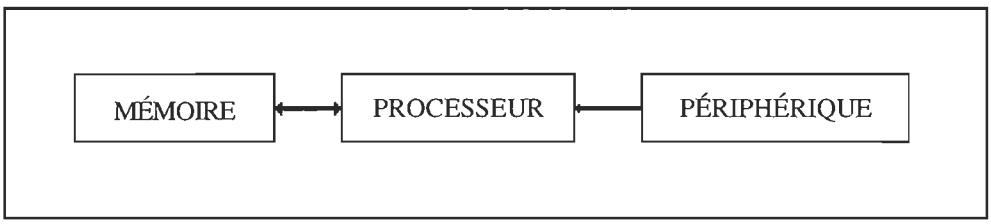


**Figure 2.5. Structure à deux bus**

Une autre structure possible est la structure à bus unique. Tel que présenté à la figure 2.6, le processeur utilise alors un bus unique pour communiquer avec tous les composants du système.

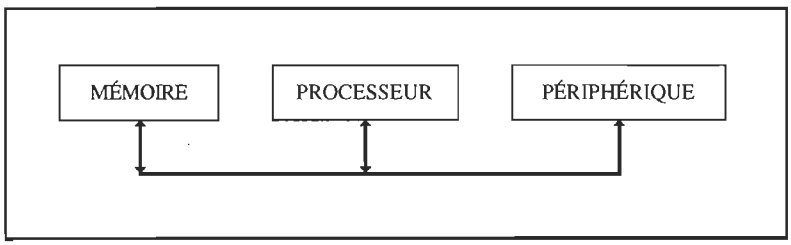


**Figure 2.6. Structure à bus unique**

## 2.3 Le chemin de données

Lors de l'exécution des instructions, les diverses manipulations de contrôles et de

séquencements ainsi que la circulation des données constituent l'essentiel du chemin de données. Telle que représentée à la figure 2.7, une telle structure utilise 8 unités principales : l'UAL, 3 multiplexeurs, un banc de registres, un démultiplexeur, une unité d'horloge et une unité de contrôle.

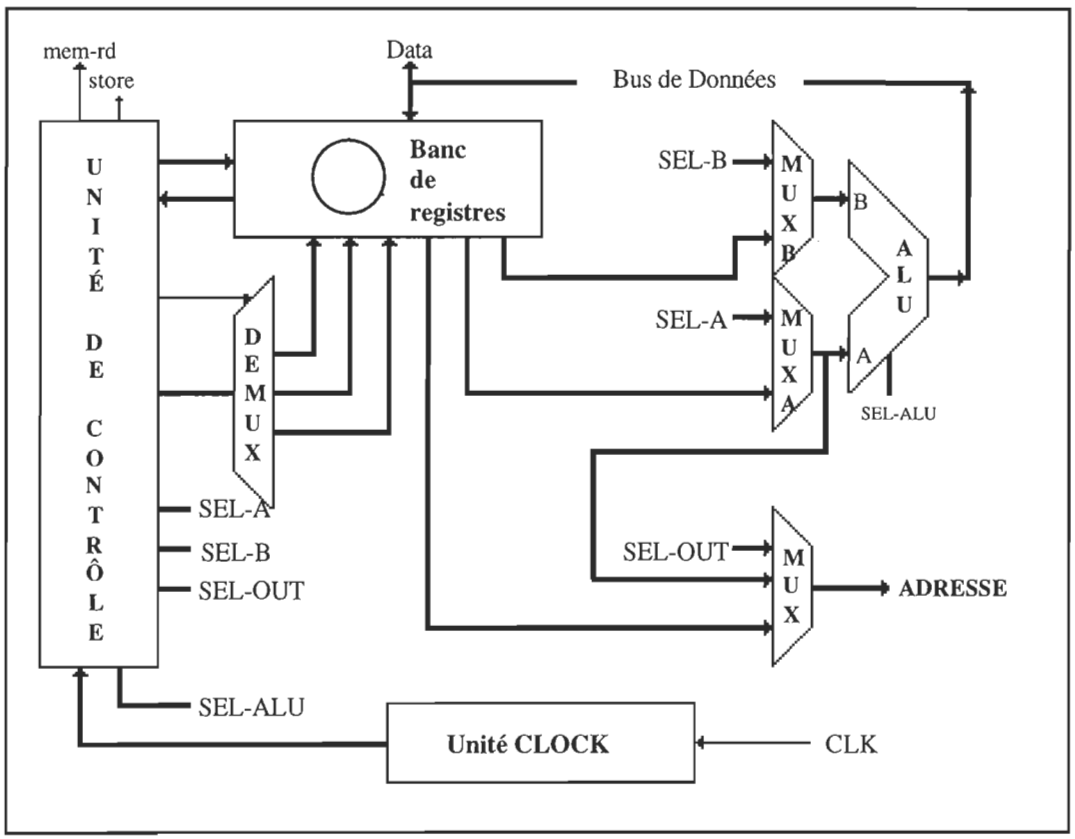


**Figure 2.7. Diagramme bloc du processeur**

L'unité arithmétique et logique (l'UAL) utilisée par le processeur est de 16 bits et elle est représentée à la figure 2.8. Elle réalise les diverses opérations arithmétiques et logiques. Généralement, les résultats de celle-ci peuvent modifier les drapeaux N, C et Z du registre SCR. Elle a un fonctionnement parallèle et elle utilise 3 bus de données (A, B et sortie) et un multiplexeur qui sélectionne un des résultats. Les multiplexeurs «muxa», «muxb» et «mux» sont utilisés pour sélectionner le contenu des registres aux entrées A et B de l'UAL ainsi que le contenu du registre utilisé comme adresse. Le banc de registres contient tous les registres du processeur.

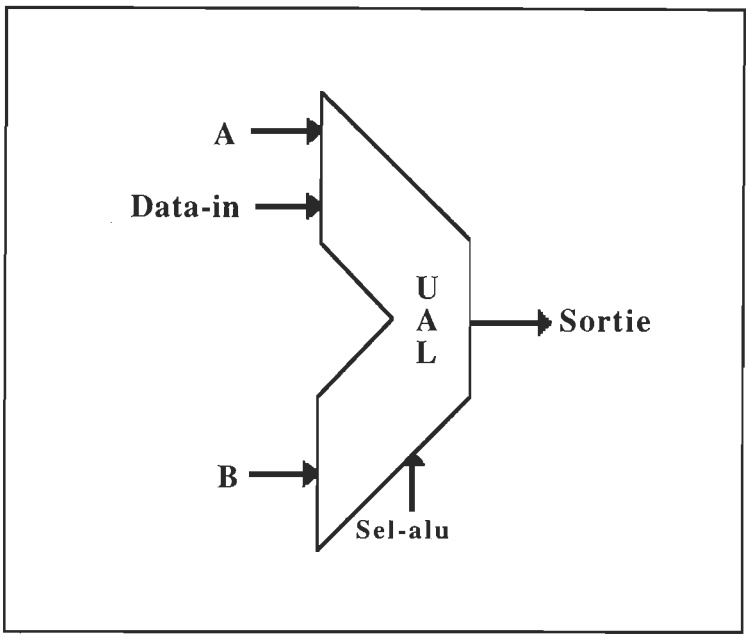


**Figure 2.8. Unité arithmétique et logique (UAL)**

Le démultiplexeur permet d'envoyer un signal d'horloge au registre sélectionné. L'unité «CLOCK» produit les différentes horloges (CLK, CLK1 et FETCH) servant aux contrôles de séquencements indispensables pour l'exécution des instructions. L'unité de contrôle sélectionne la lecture, le décodage et l'exécution des instructions en réalisant diverses manipulations de contrôles dépendant de l'instruction, de l'état du processeur ainsi que des horloges.

### 2.3.1 Exécution des instructions floues, arithmétiques et logiques

La figure 2.9 illustre le chronogramme des instructions floues, arithmétiques et logiques. Dans un premier temps, le processeur lit l'instruction existante dans la mémoire. La demande d'accès en lecture est effectuée par le signal «mem_rd» à 1 et l'horloge «load_ir» permet au processeur d'effectuer la lecture. Puisque IR1 est de 32 bits, la lecture s'effectue en deux temps. Premièrement, le processeur lit la partie haute d'IR1 (les bits 31 à 16) et il incrémente le PC à l'aide du signal «inc_pc». Ensuite, il lit la partie basse d'IR1 (les bits 15 à 0) et il incrémente de nouveau le PC qui pointe alors

vers la prochaine instruction à décoder. Ensuite, le signal «set_pipe» permet de transférer le contenu d'IR1 vers IR2. Cette opération complète le décodage de l'instruction et elle coïncide avec le début de la phase d'exécution proprement dite de l'instruction floue. Dans le même temps, le processeur commence à lire une nouvelle instruction.

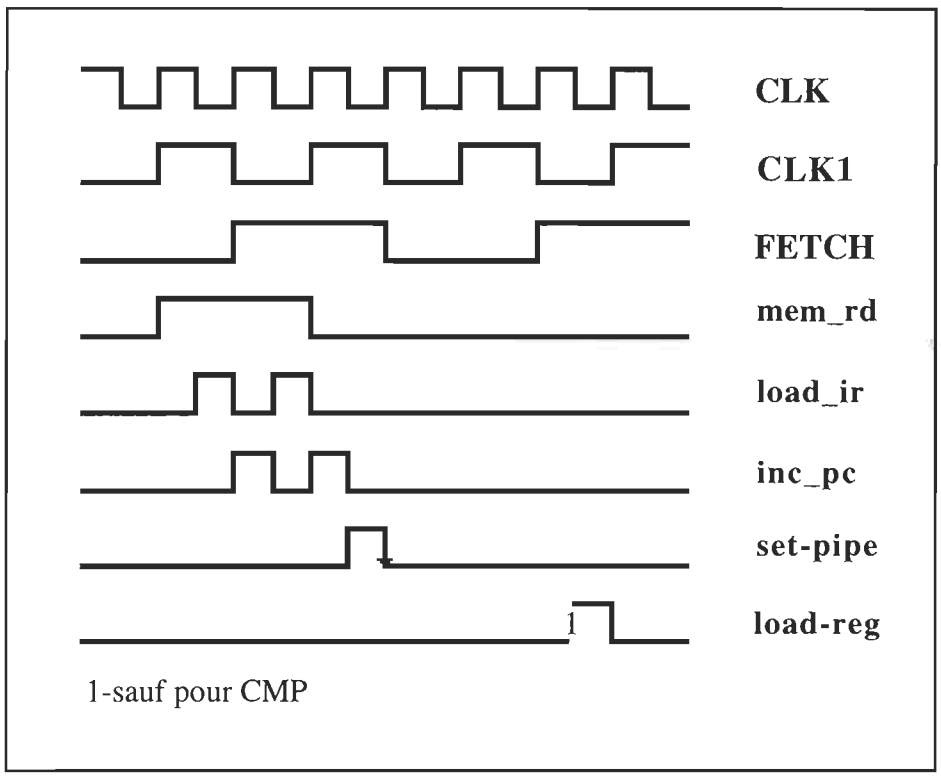


**Figure 2.9. Chronogramme des instructions floues, arithmétiques et logiques**

Le code d'opération (les bits 31 à 27 d'IR2) permet de sélectionner l'opération logique ou arithmétique à exécuter. Si le bit 26 est à 1 (mode immédiat), alors une donnée immédiate est mise à l'entrée B de l'UAL. Sinon, le contenu du registre de sources $R_b$ sélectionné (registre dont l'adresse est contenue dans les bits 15 à 11 d'IR2) est mis à cette entrée. Dans le même temps, le contenu du registre de sources $R_a$ (registre dont l'adresse apparaît dans les bits 20 à 16 d'IR2) est mis à l'entrée A de l'UAL. Le résultat est transféré dans le registre de destination $R_d$ (registre indiqué par les bits 25 à 21

d'IR2) en utilisant le signal démultiplexé «load_reg». Le signal «load_reg» permet aussi, pour les instructions arithmétiques et logiques, de modifier l'état du processeur en modifiant les drapeaux N, C et Z. Le registre de destination ne peut pas être IR, SCR ou PC.

### 2.3.2 Exécution des instructions JMP, BEQ, BCS et BMI

Le décodage de ces instructions est effectué de la même manière que pour les instructions arithmétiques et logiques. Le chronogramme de ces instructions est présenté à la figure 2.10.

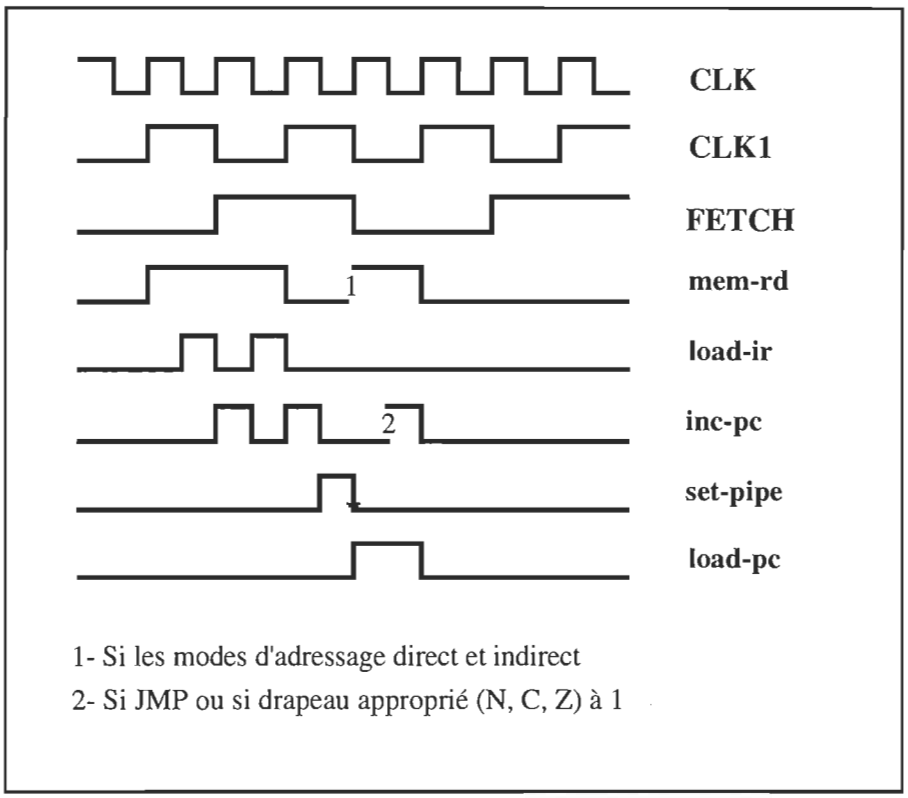


**Figure 2.10. Chronogramme des instructions JMP, BEQ, BMI et BCS**

Dans le mode d'adressage immédiat (bit 26 d'IR2 à 1), la valeur immédiate (les bits 15 à 0 d'IR2) représentant l'opérande est mise sur l'entrée B de l'UAL qui, par la suite, met cette valeur sur le bus de données. Ensuite, les signaux «load_pc» et «inc_pc»

permettent de transférer cette valeur dans le PC et de modifier son contenu en effectuant ainsi un saut. Dans le mode registre (les bits 18 à 16 d'IR2 contiennent 1), le processeur effectue les mêmes opérations décrites précédemment sauf que le contenu du registre $R_b$ est mis sur le bus de données au lieu de la partie basse d'IR2. Dans le mode direct (les bits 18 à 16 ont la valeur 2), le processeur lit une donnée de la mémoire et l'UAL met celle-ci sur le bus de données. Ensuite, «load_pc» et «inc_pc» effectuent le branchement. Durant la lecture de la donnée, la partie basse d'IR2 est utilisée comme adresse. Le mode indirect (les bits 18 à 16 d'IR2 ont la valeur 4) est réalisé de la même manière que pour le mode direct sauf que l'adresse utilisée est le contenu du registre de sources $R_b$ au lieu de la partie basse d'IR2. Il faut cependant noter que l'unité de contrôle autorise le branchement (transition sur «inc_pc») seulement lorsqu'on exécute l'instruction JMP ou lorsque la condition testée est vraie.

### 2.3.3 Exécution des interruptions matérielles et des instructions JSR$_P$ et SWI$_N$

Le décodage des instructions JSR$_P$ et SWI$_N$ se fait encore une fois de la même manière que pour les instructions arithmétiques et logiques. Toutefois, telle que représentée à la figure 2.11, l'unité de contrôle génère un signal supplémentaire «inc_fen» qui permet d'incrémenter PTF et de déplacer ainsi la roue en rendant courante une nouvelle fenêtre tout en sauvegardant l'adresse de retour.

L'exécution de JSR$_P$ est similaire à celle de l'instruction JMP sauf que l'unité de contrôle génère trois signaux supplémentaires : «jsr», «sel» et «load_reg». Ceux-ci sont utilisés par la roue de fenêtres pour faciliter le passage de paramètres. Lorsque les signaux «sel» et «jsr» sont à 1, les valeurs des registres $F_0$ et $F_1$ de l'ancienne fenêtre sont mises aux entrées correspondantes de la fenêtre courante. Ensuite, tout en faisant le branchement, le contenu de ces registres est transféré dans la nouvelle fenêtre par le signal «load_reg». Ce transfert est effectif seulement si le bit 20 est à 1 pour $F_0$ et si le bit 21 est à 1 pour $F_1$. Le contenu de ces deux bits est indiqué par la lettre P dans l'instruction JSR$_P$. Par exemple, dans le cas de JSR$_2$, le 2 signifie que seulement le registre $F_1$ est utilisé pour passer un paramètre à une procédure.

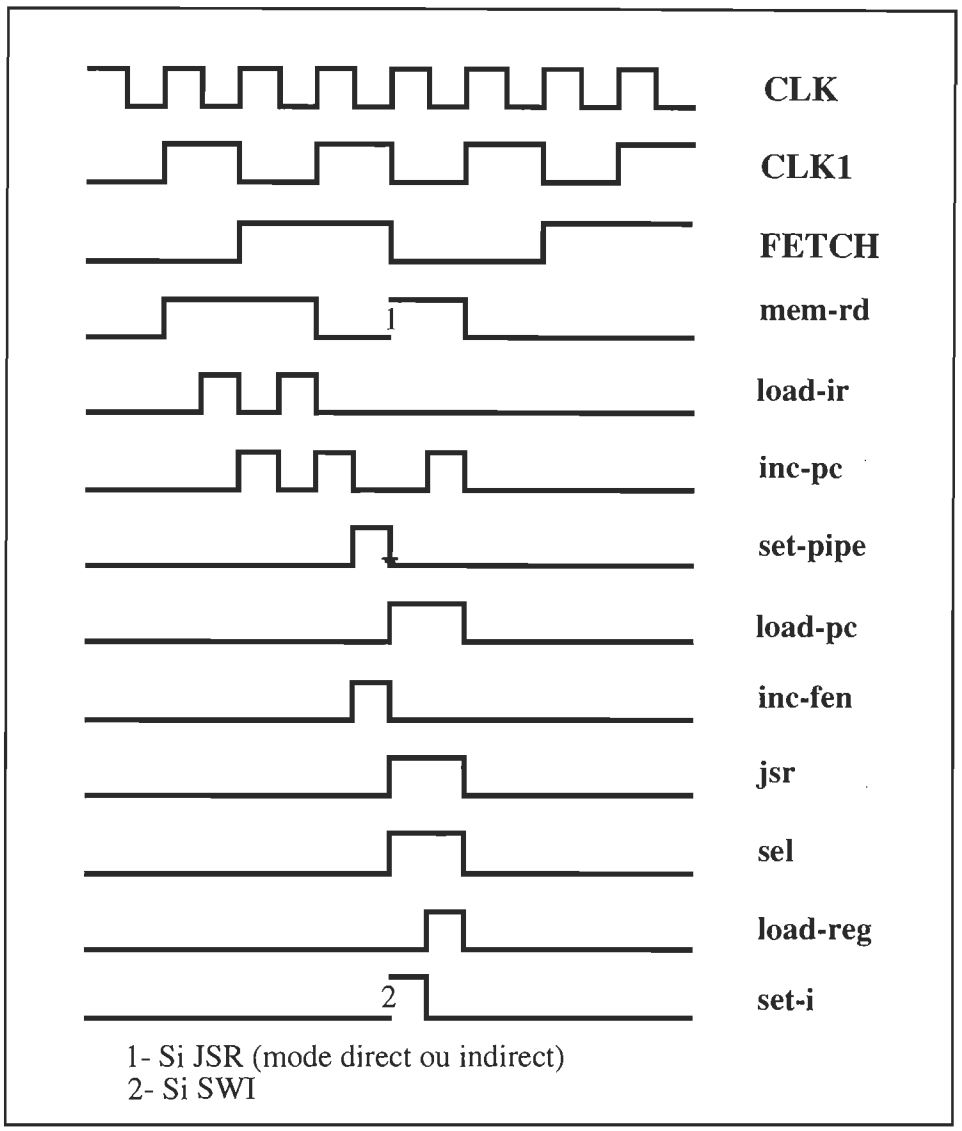


**Figure 2.11. Chronogramme des interruptions matérielles et logicielles et de l'instruction $JSR_P$**

L'exécution de $SWI_N$ est similaire à celle de JSR0, sauf qu'on modifie l'état du processeur et on fait un branchement à l'adresse du vecteur d'interruptions logicielles SWIVEC. Ces vecteurs sont contenus dans les bits 17 et 16 d'IR2 et ils sont représentés par la lettre $_N$ dans $SWI_N$. Lors de son exécution, le signal «set_i» force le drapeau I à 1. Les bits 2 et 1 du PC prennent la valeur des bits 17 et 16 d'IR2, le bit 3

du PC est mis à 1 et tous les autres bits de ce dernier sont mis à 0. Le branchement à SWIVEC est ainsi effectué.

L'exécution des interruptions matérielles suit le même principe que celle de l'instruction $SWI_N$, sauf que le vecteur d'interruptions SWIVEC est remplacé par IRQVEC (adresse 02) pour IRQ et NMIVEC (adresse 04) pour NMI. Lorsque le processeur répond à une interruption matérielle, la lecture des instructions est interrompue et le processeur effectue un saut à l'adresse du vecteur d'interruption. L'interruption IRQ est autorisée seulement si les drapeaux I et IRQ sont à 0 et à 1 respectivement. L'interruption NMI est toujours autorisée indépendamment des drapeaux I et IRQ.

Pour une profondeur d'imbrications supérieure à 8, la roue génère une exception qui, tout en bloquant le processeur, permet de sauvegarder le contenu des registres de la fenêtre courante dans la pile pointée par le registre SP. Tel que montré à la figure 2.12, après l'écriture dans la pile du contenu d'un registre, le registre SP est incrémenté.

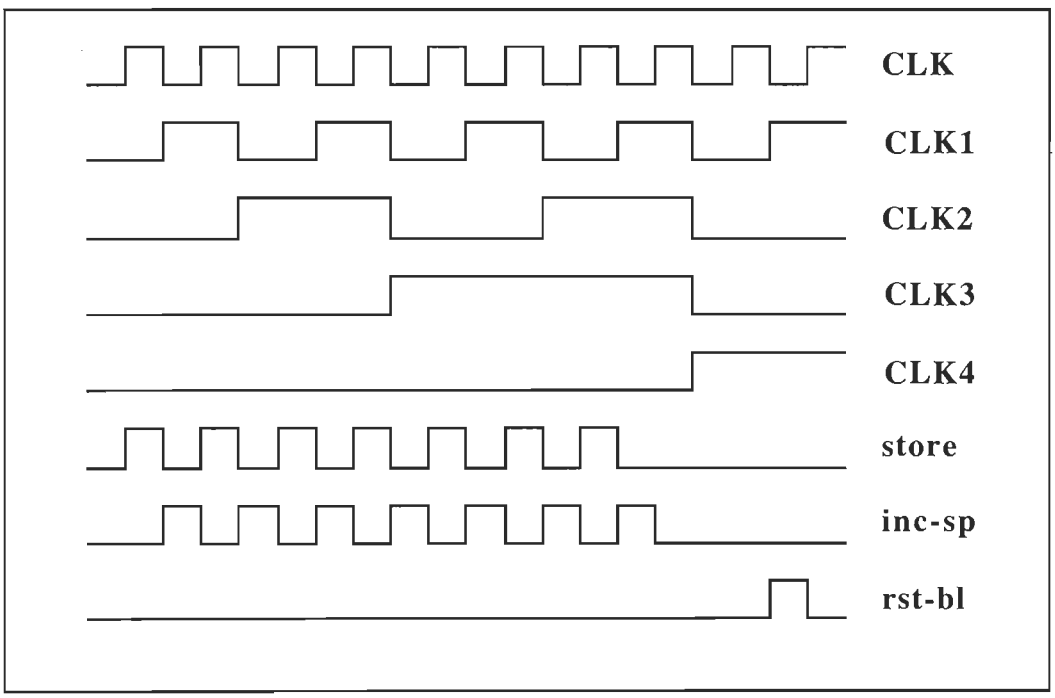


**Figure 2.12. Sauvegarde du contenu des registres d'une fenêtre**

### 2.3.4 Exécution des instructions RTSp et RTI

Le décodage de ces instructions est similaire à celui de JSRp, sauf que l'utilisation du signal «dec» permet de décrémenter le registre PTF et de rendre active une nouvelle fenêtre. Le processeur effectue ainsi un retour de sous-routine tout en rétablissant l'adresse de retour. En suivant le même principe que JSRp, l'instruction RTSp peut passer des paramètres (c'est-à-dire les registres $F_0$ et $F_1$ de l'ancienne fenêtre) à la nouvelle fenêtre. Quant à l'instruction RTI, son exécution est similaire à $RTS_0$, sauf qu'elle force aussi le drapeau I à 0. Le chronogramme de ces instructions est présenté à la figure 2.13.

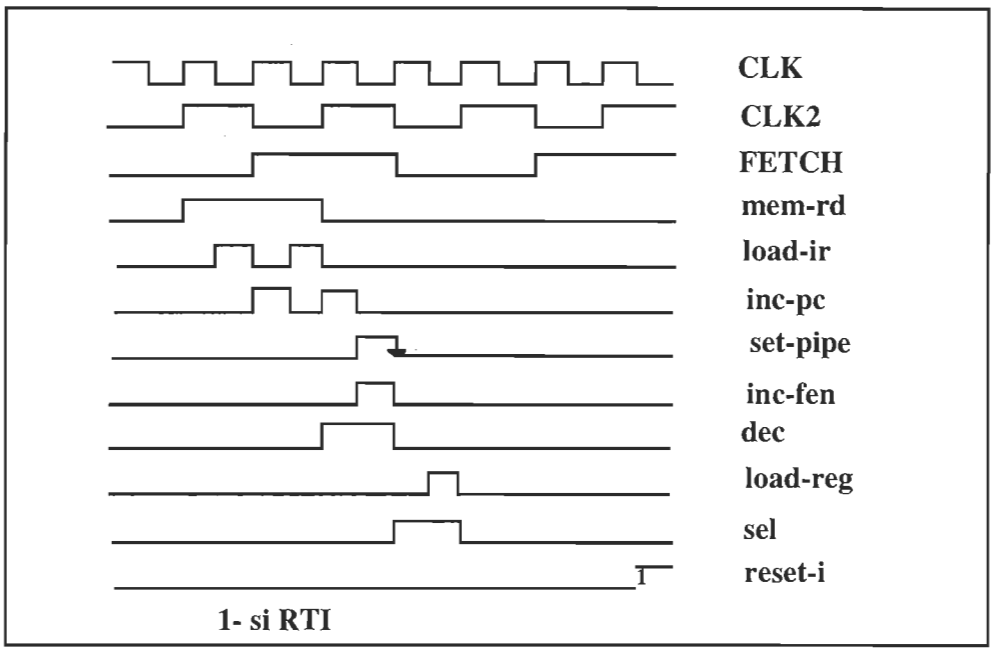


**Figure 2.13. Chronogramme des instructions RTSp et RTI**

Similairement à l'instruction JSRp, les imbrications supérieures à 8 permettent à la roue de générer une exception qui rétablit le contenu des registres de la fenêtre sauvegardée dans la pile. Pour ce faire, le registre SP est décrémenté avant la lecture du contenu de chaque registre de cette fenêtre et, pour accéder à l'ancienne fenêtre, on décrémente le registre PTF. La figure 2.14 présente le chronogramme qui permet le rétablissement du contenu des registres d'une fenêtre.

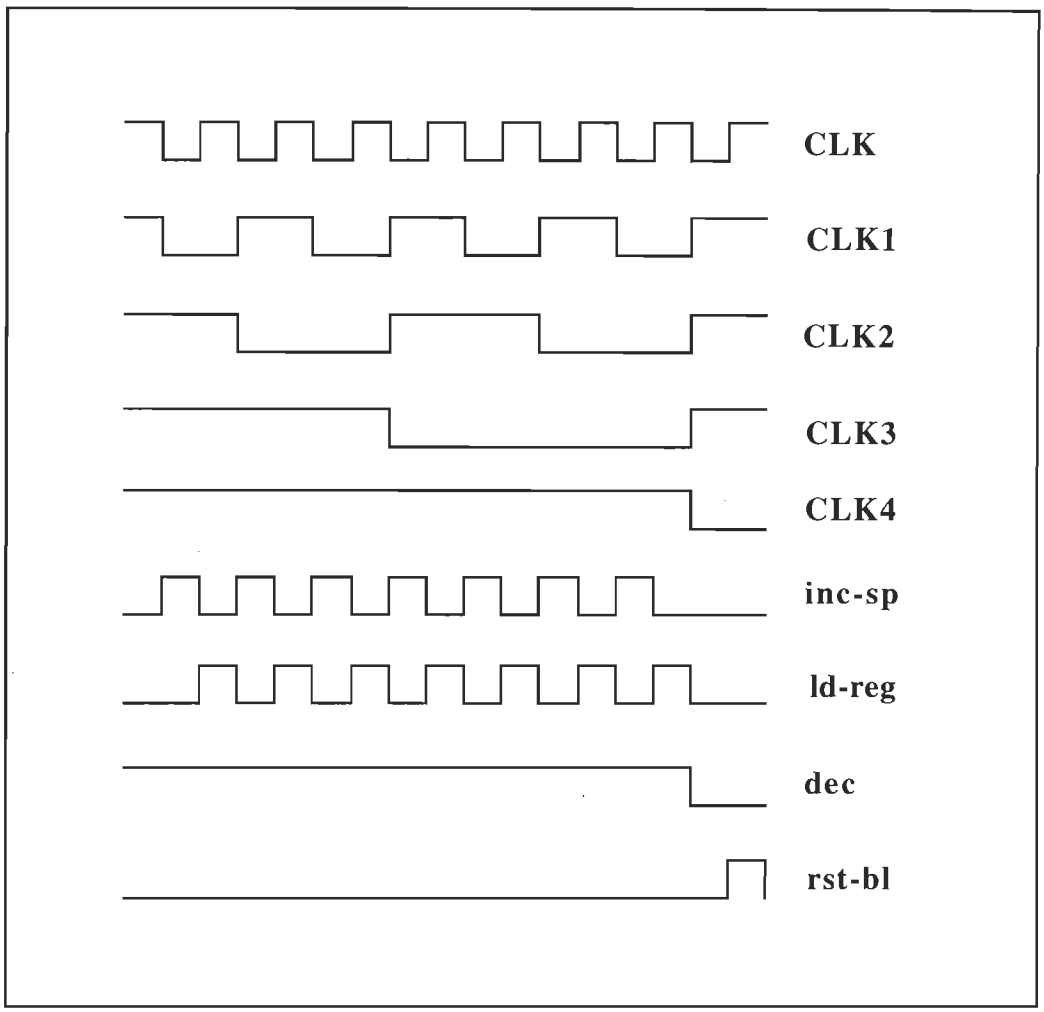


**Figure 2.14. Rétablissement du contenu des registres d'une fenêtre**

### 2.3.5 Exécution des instructions d'accès à la mémoire

Le décodage des instructions LDR et STR est identique à celui des instructions logiques. Quant à l'exécution de l'instruction LDR, elle est comparable à celle de l'instruction JMP. Puisqu'elle ne vise pas à faire un saut alors le registre de destination $R_d$, contrairement à l'instruction JMP, n'est pas le PC, mais plutôt un registre quelconque ( sauf IR et PC). En ce sens, le signal «load_pc» est remplacé par «load_ptr» et «inc_pc» par «load_reg». La figure 2.15 représente le chronogramme des

instructions d'accès à la mémoire.

L'instruction STR permet d'écrire le contenu d'un registre à la mémoire à partir de l'UAL qui met le contenu du registre de sources $R_d$ sur le bus de données et utilise le signal «store» pour entamer l'écriture. Dans le mode direct, la partie basse d'IR2 est utilisée comme adresse et dans le mode d'adressage indirect, le contenu du registre de sources $R_a$ sert d'adresse.

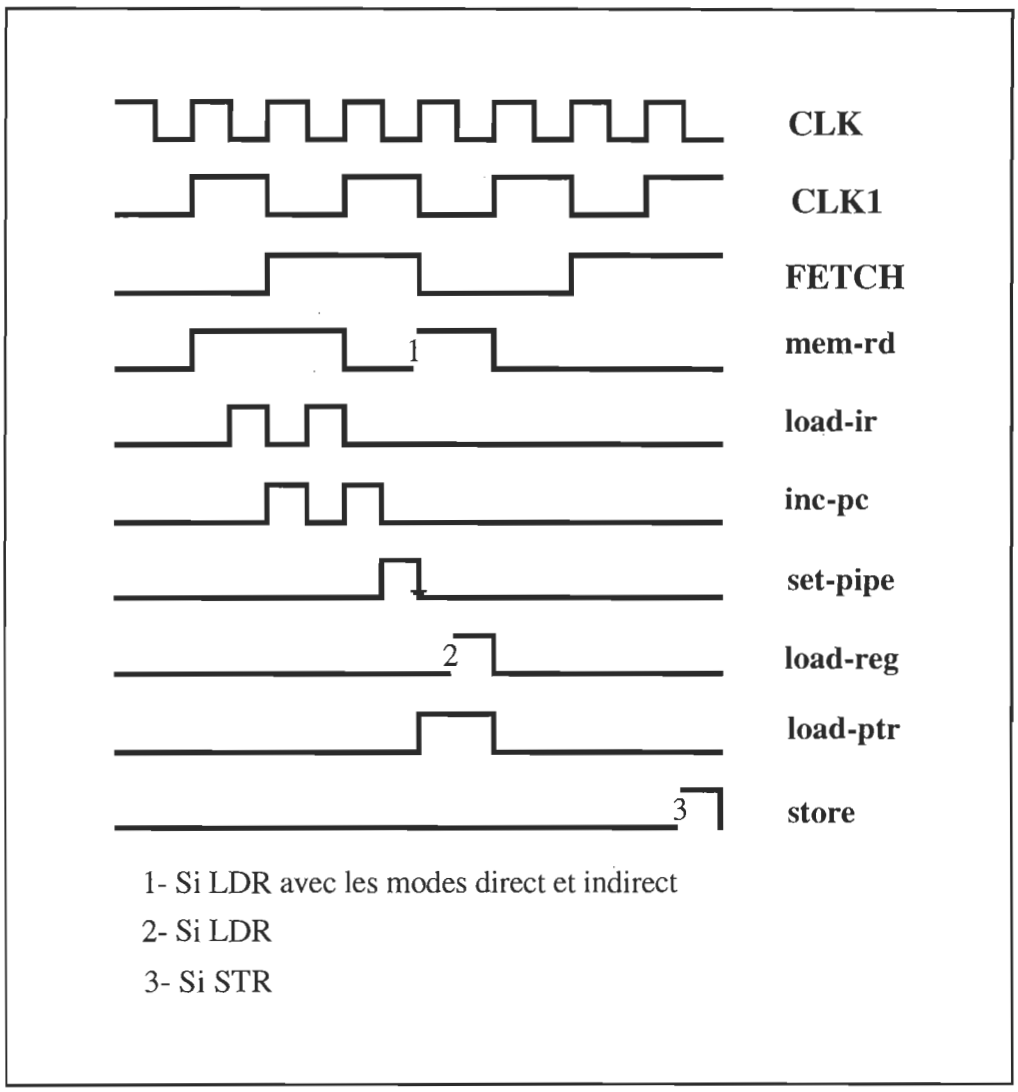


**Figure 2.15. Chronogramme des instructions d'accès à la mémoire**

### 2.3.6 Exécution des instructions NOP, SEI et CLI

Le décodage de ces instructions est identique à celui des instructions arithmétiques et logiques. Lors de l'exécution de l'instruction NOP, le processeur ne fait aucune opération particulière et avance seulement le PC à l'adresse de l'instruction suivante. L'instruction SEI force le drapeau I à 1, l'instruction CLI force ce drapeau à 0. La figure 2.16 représente le chronogramme de ces instructions.

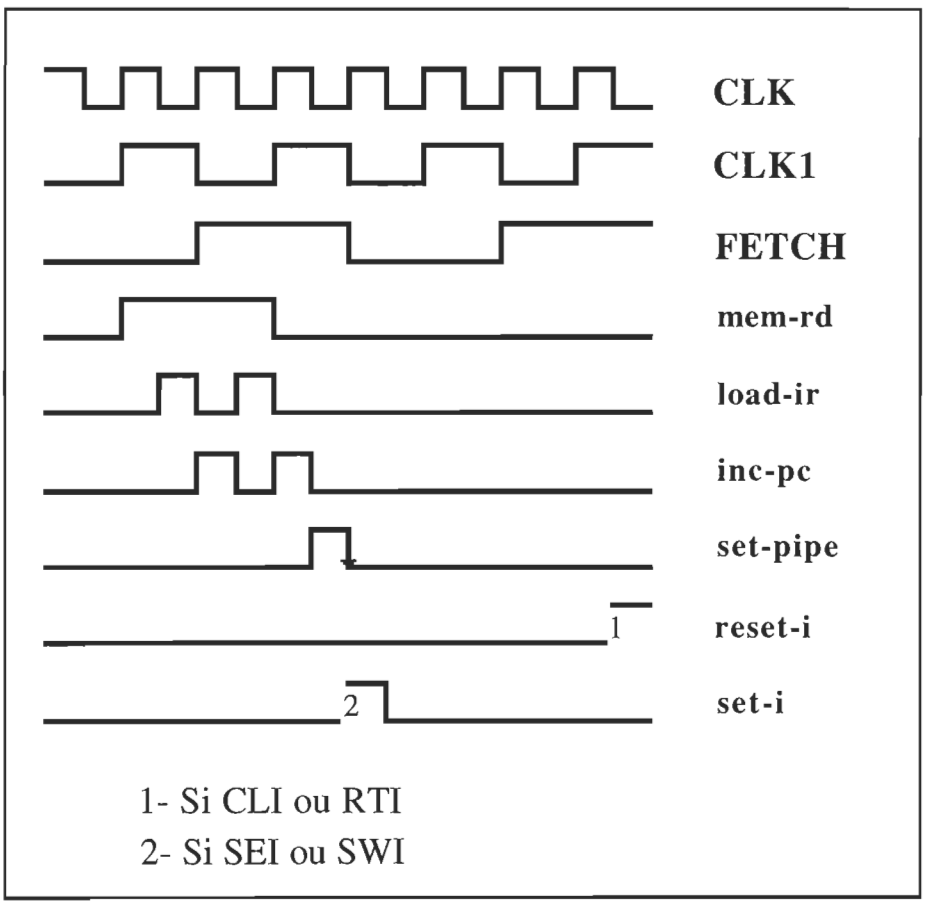


**Figure 2.16. Chronogramme des instructions NOP, SEI et CLI**

L'utilisation d'un pipeline à deux étages a permis d'éliminer toutes les sortes d'aléas en synchronisant adéquatement tous les signaux de contrôles de séquencements servant à la lecture, au décodage et à l'exécution des instructions.

# CHAPITRE III

## CONCEPTION LOGIQUE

La conception logique du processeur consiste à décrire celui-ci avec un outil de description de matériels afin de pouvoir, éventuellement, le matérialiser dans un circuit intégré. Cette description peut se faire avec un dessin de masque, avec un schéma logique, avec une table de vérité, avec des équations logiques, avec un langage évolué, ou encore avec un langage de description de matériels.

Dans ce mémoire, le processeur a été décrit avec le langage de description de matériels VHDL. Étant donné que ce langage est normalisé par IEEE ( «*Institut of Electrical and Electronics Engineers*»), plusieurs outils d'aide CAO (Conception Assistée par Ordinateurs) compatibles avec celui-ci sont actuellement disponibles. Il existe aussi plusieurs bibliothèques de modèles VHDL pouvant faciliter la conception et la simulation d'un système numérique. Le langage VHDL permet d'obtenir une description hiérarchisée composée de petites unités assemblées. De plus, tel que présenté à la figure 3.1, son haut niveau d'abstraction facilite la compréhension, la conception logique ainsi que le développement d'un système aisément modifiable.

Ces principaux niveaux d'abstraction sont la spécification d'entité, l'architecture et la configuration (Air, 1990). VHDL permet donc de décrire un système en faisant abstraction des niveaux inférieurs, en créant sa spécification d'entité, son architecture et, finalement, sa configuration. La spécification d'entité permet de spécifier les entrées et les sorties d'un système numérique. L'architecture permet de décrire le comportement ou la structure d'un système, ou même de mélanger les deux dans une description hybride. La configuration définit les liens physiques entre les composants dans des architectures et des entités spécifiques.

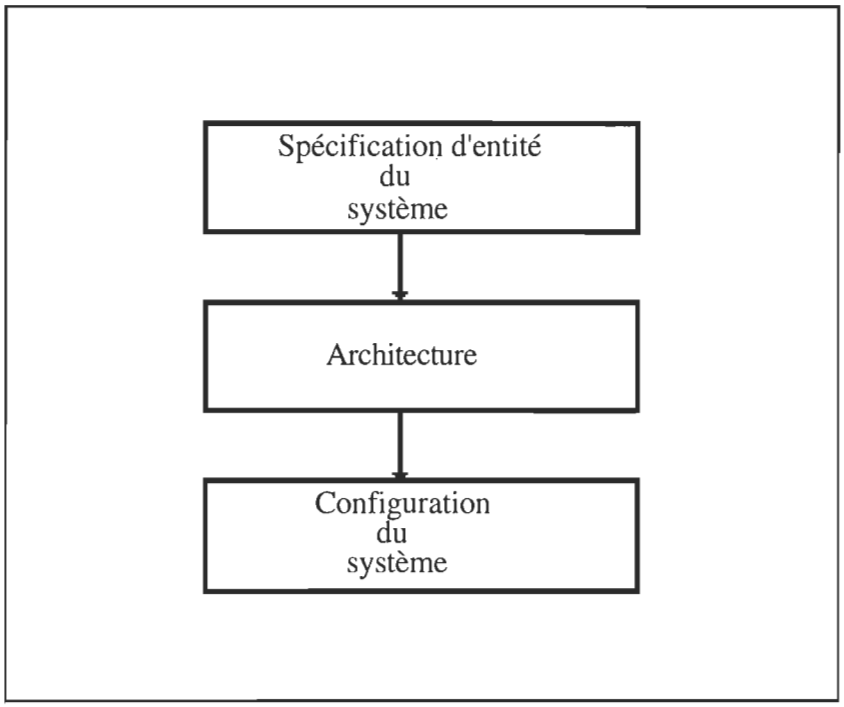


**Figure 3.1. Niveaux d'abstraction**

Telle que représentée à la figure 3.2, la méthodologie adoptée est composée de cinq étapes :

Étape 1 - Définition des spécifications matérielles du processeur;

Étape 2 - Partitionnement du système. Le système est partitionné en plusieurs unités qui sont, à leur tour d'une manière hiérarchique, partitionnées en plusieurs sous-unités;

Étape 3 - Création de modèles VHDL. Un modèle VHDL est créé pour chaque sous-unité.

Étape 4 - Compilation des modèles. Le modèle est compilé dans l'environnement d'aide à la conception CAO de Mentor Graphics. En cas d'erreurs, il faut corriger la syntaxe du code VHDL et il faut recompiler à nouveau.

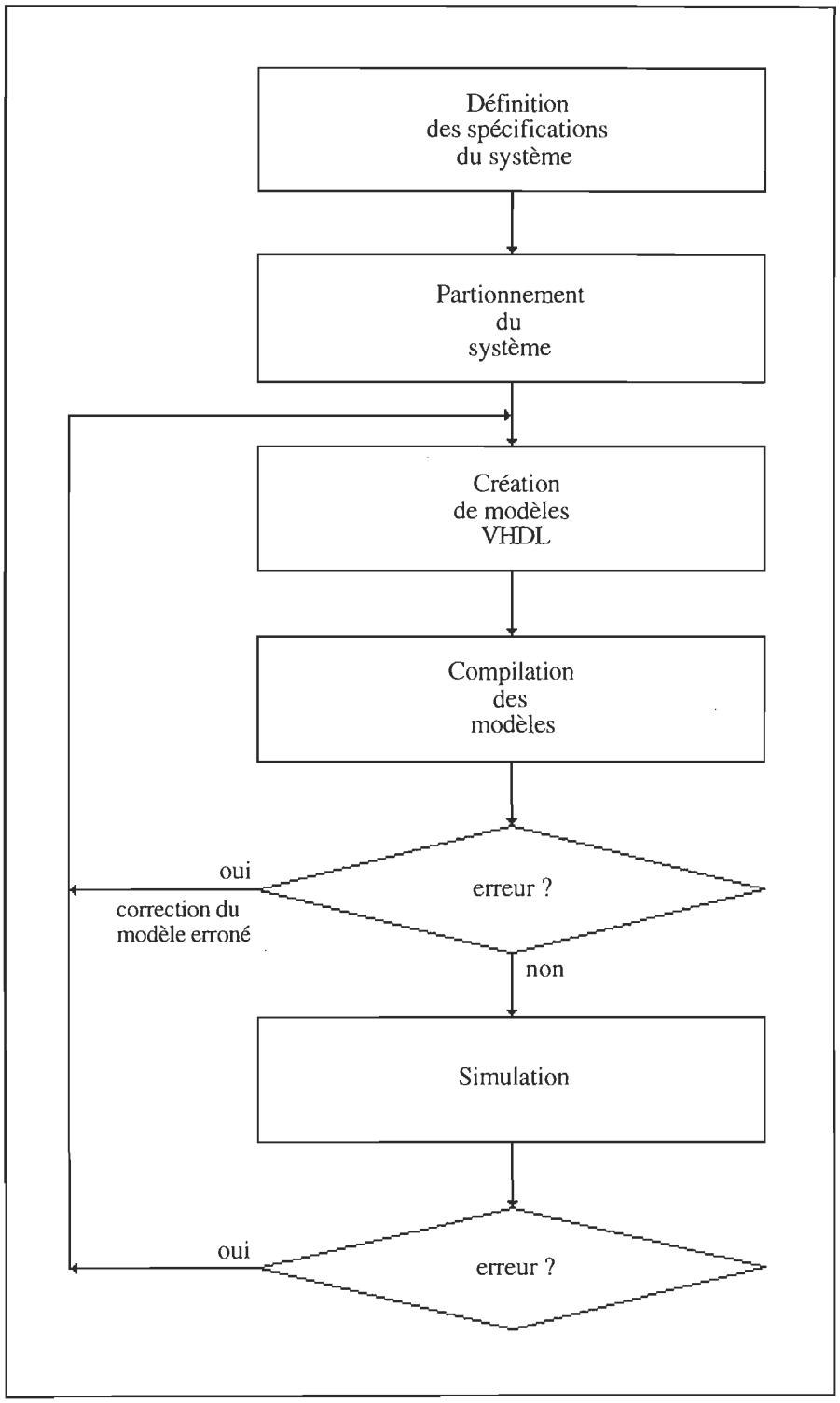


**Figure 3.2. Étapes de conception logique du processeur**

Étape 5 - Simulation. Une fois la compilation réussie, il faut simuler l'unité avec le logiciel QUICKSIM de Mentor Graphics. Si les résultats ne sont pas satisfaisants, il faut revenir à l'étape 2 afin de modifier le modèle VHDL. Après une simulation réussie, il faut revenir à l'étape 3 afin de créer et de vérifier les modèles des autres sous-unités. Ces opérations sont répétées jusqu'à ce qu'un modèle complet et fonctionnel du processeur soit obtenu.

## 3.1 La description logique du processeur

La description logique du processeur doit définir ses caractéristiques fonctionnelles tout en permettant l'exécution des différentes instructions, ainsi que le traitement des interruptions matérielles. Cette description est effectuée avec le langage de description des matériels VHDL.

Les 8 unités présentées à la figure 2.7 sont décrites pour établir un premier niveau hiérarchique du processeur. Étant donné que le processeur a été mis en oeuvre avec des circuits FPGAs de la famille XILINX-4000, il faut respecter les caractéristiques matérielles de ces circuits. Par exemple, le nombre de portes logiques disponibles, le nombre d'entrées et de sorties, le nombre de registres ainsi que le nombre de tampons à trois états sont limités. Donc, la description logique, tout en recherchant une rapidité de fonctionnement optimale du circuit, doit aussi faciliter la matérialisation de ce même circuit dans le circuit FPGA choisi.

### 3.1.1 Les multiplexeurs

Les multiplexeurs permettent de choisir un chemin de données parmi plusieurs, au moyen d'un code de sélection identifiant la donnée. Par exemple, on peut avoir un multiplexeur de $2^n$ bits de données ( V = d(0), d(1), .... d(n-1) ), avec un code de sélection S de n bits et une sortie Z. Cette dernière est obtenue par l'équation suivante :

$$Z = V(S). \qquad 3.1$$

Une telle structure s'obtient généralement en utilisant la logique combinatoire et, afin de réduire le coût matériel de ces circuits, il est possible d'utiliser des tampons à trois états qui possèdent un état de haute impédance. En connectant ensemble plusieurs sorties de ces tampons, on obtient un multiplexeur qui transfère la donnée sélectionnée sur le bus de sortie. Les multiplexeurs «muxa» et «muxb» (voir fig. 2.7) utilisent une structure mixte, c'est-à-dire un mélange de multiplexeurs avec des portes logiques et de multiplexeurs avec des tampons à trois états. Cette structure mixte permet d'obtenir une surface optimale pour ces circuits tout en limitant le nombre de tampons à trois états utilisés.

Les circuits «muxa» et «muxb» permettent donc de sélectionner le contenu d'un des 32 registres utilisables par une instruction, au moyen des codes de sélection de 5 bits «sel».Quant au troisième multiplexeur («mux» ou «muxout»), il permet de sélectionner le contenu du registre utilisé comme adresse et il est constitué par des tampons à trois états. Pour le mode d'adressage indirect, «muxout» transfère le contenu du registre sélectionné par «muxa» à sa sortie. Les modèles VHDL de ces trois multiplexeurs sont présentés aux tableaux 3.1, 3.2 et 3.3 et la simulation de l'unité «muxout», à la figure 3.3.

**TABLEAU 3.1**
**Modèle VHDL du multiplexeur «muxout»**

| **Unité «muxout»** |
|---|
| LIBRARY MGC_PORTABLE;<br>USE MGC_PORTABLE_LOGIC.ALL;<br>USE MGC_PORTABLE_RELATIONS.ALL;<br>entity muxout is port<br>(fetch, clk1, dir, ind, bloc : in qsim_state;<br>ir, pc, sp, ma : in qsim_state_vector(15 downto 0);<br>adresse : out qsim_state_resolved_x_vector(15 downto 0) bus<br>);<br>end muxout;<br>architecture comp of muxout is |

**Unité «muxout» (suite)**

```
signal s1          : qsim_state :='0';
signal a           : qsim_state_vector(1 downto 0) :="00";
begin

    s1<= fetch xor clk1;
    process(s1, dir, ind, bloc)
    begin
        if (bloc='1') then a <= "00";
          elsif ( (s1='1') or ((dir='0') and (ind='0')) ) then a <= "01";
            elsif ( (s1='0') and (dir='1') ) then a <= "10";
              else a <= "11";
        end if;
    end process;
    a0 : block(a="00")
      begin
          adresse <= guarded qsim_state_resolved_x_vector(sp);
       end block;
    a1 : block(a="01")
      begin
          adresse <= guarded qsim_state_resolved_x_vector(pc);
       end block;
    a2 : block(a="10")
      begin
          adresse <= guarded qsim_state_resolved_x_vector(ir);
       end block;
    a3 : block(a="11")
      begin
          adresse <= guarded qsim_state_resolved_x_vector(ma);
       end block;
end comp;
```

## TABLEAU 3.2

## Modèle VHDL du multiplexeur «muxa»

**Unité «muxa»**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;
entity  muxa is port
          (SEL0 : in qsim_state_vector(15 downto 11);
           SEL1 : in qsim_state_vector(20 downto 16);
           ind, dir : in qsim_state;
           D0, D1, D2, D3, D4, D5 : in qsim_state_vector(15 downto  0);
           D6, D7, D8, D9, D10, D11 : in qsim_state_vector(15 downto 0);
           D12, D13, D14, D15, D16 : in qsim_state_vector(15 downto  0);
           D17, D18, D19, D20, SP : in qsim_state_vector(15 downto 0);
           PTF, F0, F1, F2, F3 : in qsim_state_vector(15 downto 0);
           F4, F5, PC,IR, SCR : in qsim_state_vector(15 downto 0);
           O : out qsim_state_vector(15 downto 0));
end muxa;
architecture comp of muxa is
--- ce composant est un multiplexeur à 26 entrées et un code de sélection de 5 bits ----
component muxbuf26 port
                        (sel : in qsim_state_vector(4 downto 0);
                        A0, A1, A2, A3, A4, A5, A6 : in qsim_state_vector(15 downto 0);
                        A7, A8, A9, A10, A11, A12 : in qsim_state_vector(15 downto 0);
                        A13, A14, A15, A16, A17, A18 : in qsim_state_vector(15 downto 0);
                        A19, A20, A21, A22 : in qsim_state_vector(15 downto 0);
                        A23, A24, A25 : in qsim_state_vector(15 downto 0);
                        Z : qsim_state_resolved_x_vector(15 downto 0) bus
                       );
end component;
signal MODE : qsim_state :='0';
signal SEL : qsim_state_vector(4 downto  0) :="00000";
signal z : qsim_state_resolved_x_vector(15 downto 0);
for u0 : muxbuf26 use entity WORK.muxbuf26(comp);
begin
          MODE <= IND OR DIR;
          process(MODE, SEL0, SEL1)
          begin
                    if (MODE='1') then SEL <= SEL0; else SEL <= SEL1; end if;
          end process;
          u0 : muxbuf26 port map(ir, d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, d13,
                                  d14, d15, d16, d17, d18, d19, d20, sp, ptf, f0, f1, z);
          RT : process(F2, F3, F4, F5, PC, SEL, SCR(4 downto 0), z)
               begin
                         if (SEL="11010") then O <= F2;
                           elsif (SEL="11011") then O <= F3;
                                elsif (SEL="11100") then O <= F4;
                                   elsif (SEL="11101") then O <= F5;
                                      elsif (SEl="11110") then O <= PC;
                                            elsif (SEL="11111") then O(4 downto 0) <= SCR( 4 downto 0);
                                                 else O <= qsim_state_vector(z);
                         end if;
                end process;
end comp;
```

## TABLEAU 3.3

## Modèle VHDL du multiplexeur «muxb»

**Unité «muxb»**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;
entity  muxb is port
          (SEL0 : in qsim_state_vector(15 downto 11);
           SEL1 : in qsim_state_vector(20 downto 16);
           SELM : in qsim_state_vector(4 downto 0);
           BLOC : in qsim_state;
           INST  : in qsim_state_vector(31 downto 26);
           D0, D1, D2, D3, D4, D5, D6, D7, D8, D9 : in qsim_state_vector(15 downto  0);
           D10, D11, D12, D13, D14, D15, D16, D17 : in qsim_state_vector(15 downto  0);
           D18, D19, D20, SP, PTF, F0, F1, F2, F3 : in qsim_state_vector(15 downto 0);
           F4, F5, PC,IR, SCR : in qsim_state_vector(15 downto 0);
           O : out qsim_state_vector(15 downto 0));
end muxb;
architecture comp of muxb is
component muxbuf26 port
          (sel : in qsim_state_vector(4 downto 0);
           A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11 : in qsim_state_vector(15 downto 0);
           A12, A13, A14, A15, A16, A17, A18, A19, A20, A21 : in qsim_state_vector(15 downto 0);
           A22, A23, A24, A25 : in qsim_state_vector(15 downto 0);
           Z : qsim_state_resolved_x_vector(15 downto 0) BUS);
end component;
signal STORE : qsim_state :='0';
signal SEL : qsim_state_vector(4 downto  0) :="00000";
signal z : qsim_state_resolved_x_vector(15 downto 0);
for u0 : muxbuf26 use entity WORK.muxbuf26(comp);
begin
          STORE <= INST(31) and INST(30) and INST(29) and NOT INST(28) and INST(27);
          process(STORE, SEL0, SEL1, INST(26), SELM, BLOC)
          begin
                    if (BLOC='1') then SEL <= SELM;
                      elsif (INST(26)='1') then SEL <= "00000";
                        elsif (STORE='1') then SEL= SEL1;
                          else SEL <= SEL0;
                     end if;
          end process;
          u0 : muxbuf26 port map(ir, d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, d13,
                                 d14, d15, d16, d17, d18, d19, d20, sp, ptf, f0, f1, z);
          RT : process(F2, F3, F4, F5, PC, SEL, SCR(4 downto 0), z)
               begin
                         if (SEL="11010") then O <= F2;
                           elsif (SEL="11011") then O <= F3;
                                elsif (SEL="11100") then O <= F4;
                                   elsif (SEL="11101") then O <= F5;
                                      elsif (SEl="11110") then O <= PC;
                                            elsif (SEL="11111") then O(4 downto 0) <= SCR( 4 downto 0);
                                                 else O <= qsim_state_vector(z);
                          end if;
                end process;
end comp;
```

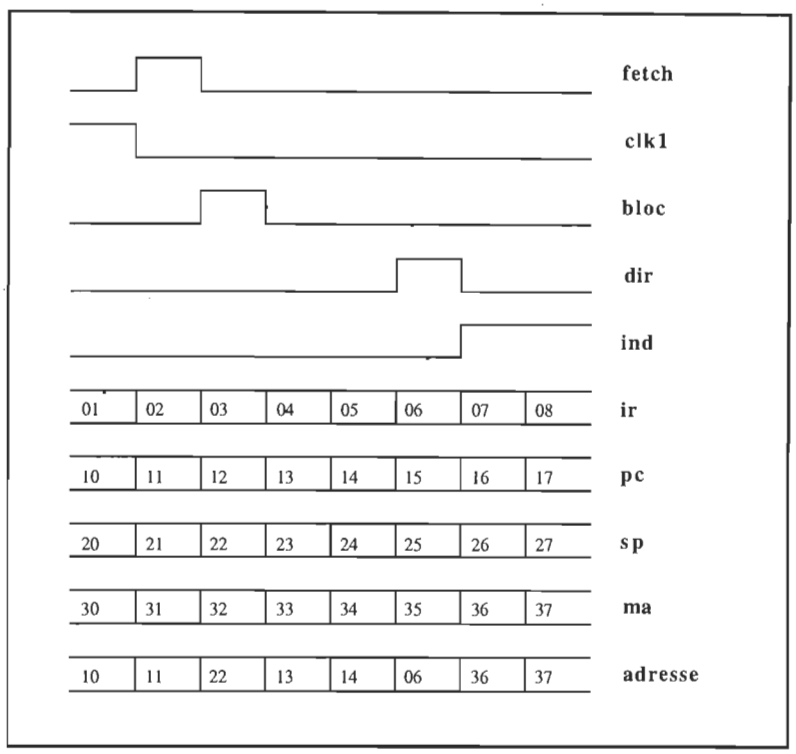


**Figure 3.3. Simulation de l'unité «muxout»**

### 3.1.2 Le démultiplexeur

Le démultiplexeur permet de transférer une donnée sur la sortie sélectionnée par un code de sélection. L'unité «demux» est utilisée comme démultiplexeur, son diagramme bloc est présenté à la figure 3.4 et son modèle VHDL, au tableau 3.4.

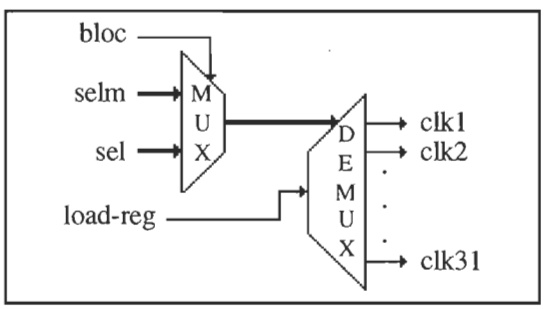


**Figure 3.4. Diagramme bloc de l'unité «demux»**

Le démultiplexeur utilisé par le processeur permet essentiellement de distribuer le signal «load_reg» vers le registre de destination sélectionné.

## TABLEAU 3.4
## Modèle VHDL du démultiplexeur «demux»

**Unité «demux»**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;
entity demux is port
        (sel, selm : in qsim_state_vector(4 downto 0);
         load_reg, para, bloc : in qsim_state;
         clk1, clk2, clk3, clk4, clk5, clk6, clk7, clk8, clk9, clk10 : out qsim_state;
         clk11, clk12, clk13, clk14, clk15, clk16, clk17, clk18 : out qsim_state;
         clk19, clk20, clk21, clk22, clk23, clkr, incpc, clk31 : out qsim_state);
end demux ;
architecture comp of demux is
signal s : qsim_state_vector(4 downto 0);
begin
        process(sel, selm, bloc)
        begin
                if (bloc='1') then s <= selm; else s <= sel; end if;
        end process;
        clk1 <= load_reg and not s(4) and not s(3) and not s(2) and not s(1) and       s(0);
        clk2 <= load_reg and not s(4) and not s(3) and not s(2) and       s(1) and not s(0);
        clk3 <= load_reg and not s(4) and not s(3) and not s(2) and       s(1) and       s(0);
        clk4 <= load_reg and not s(4) and not s(3) and       s(2) and not s(1) and not s(0);
        clk5 <= load_reg and not s(4) and not s(3) and       s(2) and not s(1) and       s(0);
        clk6 <= load_reg and not s(4) and not s(3) and       s(2) and       s(1) and not s(0);
        clk7 <= load_reg and not s(4) and not s(3) and       s(2) and       s(1) and       s(0);
        clk8 <= load_reg and not s(4) and       s(3) and not s(2) and not s(1) and not s(0);
        clk9 <= load_reg and not s(4) and       s(3) and not s(2) and not s(1) and       s(0);
        clk10<= load_reg and not s(4) and       s(3) and not s(2) and       s(1) and not s(0);
        clk11<= load_reg and not s(4) and       s(3) and not s(2) and       s(1) and       s(0);
        clk12<= load_reg and not s(4) and       s(3) and       s(2) and not s(1) and not s(0);
        clk13<= load_reg and not s(4) and       s(3) and       s(2) and not s(1) and       s(0);
        clk14<= load_reg and not s(4) and       s(3) and       s(2) and       s(1) and not s(0);
        clk15<= load_reg and not s(4) and       s(3) and       s(2) and       s(1) and       s(0);
        clk16<= load_reg and       s(4) and not s(3) and not s(2) and not s(1) and not s(0);
        clk17<= load_reg and       s(4) and not s(3) and not s(2) and not s(1) and       s(0);
        clk18<= load_reg and       s(4) and not s(3) and not s(2) and       s(1) and not s(0);
        clk19<= load_reg and       s(4) and not s(3) and not s(2) and       s(1) and       s(0);
        clk20<= load_reg and       s(4) and not s(3) and       s(2) and not s(1) and not s(0);
        clk21<= load_reg and       s(4) and not s(3) and       s(2) and not s(1) and       s(0);
        clk22<= load_reg and       s(4) and not s(3) and       s(2) and       s(1) and not s(0);
        clk23<= load_reg and       s(4) and not s(3) and       s(2) and       s(1) and       s(0);
        clkr<=(  (s(4) and s(3) and not s(2) and not s(1) and not s(0)) or (s(4) and s(3) and not s(2) and not
                  s(1) and s(0)) or (s(4) and s(3) and not s(2) and s(1) and not s(0)) or (s(4) and s(3) and not
                  s(2) and s(1) and s(0)) or (s(4) and s(3) and s(2) and not s(1) and not s(0)) or (s(4) and s(3)
                 and s(2) and not s(1) and  s(0)) or (para)) and load_reg;
        incpc<= load_reg and    s(4) and    s(3) and    s(2) and    s(1) and not  s(0);
        clk31<= load_reg and    s(4) and    s(3) and    s(2) and    s(1) and      s(0);
end comp;
```

### 3.1.3 L'unité «CLOCK»

L'unité «CLOCK» fournit les différentes horloges nécessaires au fonctionnement du processeur. Telles que présentées au tableau 3.5, ces horloges proviennent d'un compteur synchrone de deux bits qui est constitué d'un registre de deux bits, d'un circuit d'incrémentation et d'un tampon à trois états. Ce dernier permet de conserver les horloges dans un état déterminé lorsque le signal «sel» est au niveau logique 1.

**TABLEAU 3.5**
**Modèle VHDL de l'unité «CLOCK»**

**Unité «CLOCK»**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;

entity horloges  is port
          (Reset, Clk, SEL : IN QSIM_STATE;
           clk0,clk1, fetch : OUT QSIM_STATE);
end horloges;

architecture comp of horloges is

signal D0 : QSIM_STATE_vector(1 downto 0) :="00";
signal clki : qsim_state;
signal clkbuf : qsim_state_resolved_x bus;

begin

          aa : block(sel='0')
               begin
                    clkbuf <= guarded qsim_state_resolved_x(clk);
               end block;

          h : process(Clki, Reset)
              begin
                 if (Reset='1') then D0 <=  "00" ;
                  elsif ( (Clki ='1') and (Clki'event) and (Clki'last_value ='0')) then D0 <= D0 + "01";
                 end if;
              end process;
          clk1 <= D0(0);
          fetch <= D0(1);
          clk0 <= qsim_state(clkbuf);
          clki <= qsim_state(clkbuf);
end comp;
```

### 3.1.4 Le banc de registres

Le banc de registres est constitué de deux unités : l'unité «banc» et la roue de fenêtres (voir fig. 2.4). L'unité «banc» contient les registres IR, $D_0$ à $D_{20}$, SP, PTF et SCR. Le registre IR est formé de deux registres de 16 bits synchronisés sur un front montant et de deux autres registres de 16 bits synchronisés sur un front descendant. $D_0$ à $D_{20}$ sont formés par des registres de 16 bits synchronisés sur un front montant. SP et PTF sont des compteurs de 16 bits qui peuvent être incrémentés ou décrémentés de 1. SCR est constitué par trois registres synchronisés sur un front montant contenant les drapeaux «N», «C» et «Z» et de deux registres synchronisés sur un niveau logique haut contenant les drapeaux IRQ et I. Le modèles VHDL de l'unité «banc» est présenté au tableau 3.6.

**TABLEAU 3.6**
**Modèle VHDL de l'unité «banc»**

**Unité «banc»**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;
entity banc is port
        (reset, cd0, cd1, cd2, cd3, cd4, cd5, cd6, cd7, cd8, cd9, cd10, cd11, cd12, cd13 : in qsim_state;
         cd14, cd15, cd16, cd17, cd18, cd19, cd20, csp, ldsp, decsp, cptf, ldptf, decptf : in qsim_state;
         set_pipe, load_reg, set_i, reset_i, load_ir1, load_ir2, cdscr : in qsim_state;
         data_in : in qsim_state_vector(15 downto 0);
           alu_out : in qsim_state_vector(16 downto 0);
           ir1, ir2 : out qsim_state_vector(31 downto 0);
          d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10,d11,d12,d13 :out qsim_state_vector(15 downto 0);
           d14, d15, d16, d17, d18, d19, d20, sp, ptf, scr : out qsim_state_vector(15 downto 0);
           n,c, z,i, irq : out qsim_state);
end banc ;
architecture comp of banc is
component drapeau port( clk, load_reg, reset, set_i, reset_i : in qsim_state;
                                alu_out : in qsim_state_vector(16 downto 0);
                                ir2 : in qsim_state_vector(31 downto 28);
                                data_in : in qsim_state_vector(4 downto 0);
                                 n, c, z, i, irq : out qsim_state);
end component ;
component reg16 port(  Reset, Clk : in qsim_state;
                              Data_in : in qsim_state_vector( 15 downto 0);
                              Data_out : out qsim_state_vector( 15 downto 0 ));
end component;
```

**Unité «banc» (suite)**

```
component compteur port( Reset, Clk, dec, Ld : in qsim_state;
                         Data_in : in qsim_state_vector( 15 downto 0);
                         Data_out : out qsim_state_vector(15 downto 0));
end component;
component regir port (Reset, Clk, Clk1, Clk2 : in qsim_state;
                      Data_in : in  qsim_state_vector( 15 downto 0 );
                      ir1,ir2 : out qsim_state_vector( 31 downto 0 ));
end component;
signal Dra : qsim_state_vector(31 downto 0);
signal i1, irq1, c1, n1,z1   : qsim_state :='0';
FOR U1, U2, U3, U4, U5, U6, U7, U8, U9, U10, U20 : reg16 use entity WORK.reg16(comp);
FOR U11, U12, U13, U14, U15, U16, U17, U18, U19, U21 : reg16 use entity
WORK.reg16(comp);
FOR U22, U23 : compteur use entity WORK.compteur(comp);
FOR UO : regir use entity WORK.regir(comp);
FOR U31 : drapeau use entity WORK.drapeau(comp);

begin
        UO : regir  port map (reset, load_ir1, load_ir2, set_pipe, Data_in, ir1, dra);
        U1 :  reg16 port map(reset, cd0, alu_out(15 downto 0), D0);
        U2 :  reg16 port map(reset, cd1, alu_out(15 downto 0), D1);
        U3 :  reg16 port map(reset, cd2, alu_out(15 downto 0), D2);
        U4 :  reg16 port map(reset, cd3, alu_out(15 downto 0), D3);
        U5 :  reg16 port map(reset, cd4, alu_out(15 downto 0), D4);
        U6 :  reg16 port map(reset, cd5, alu_out(15 downto 0), D5);
        U7 :  reg16 port map(reset, cd6, alu_out(15 downto 0), D6);
        U8 :  reg16 port map(reset, cd7, alu_out(15 downto 0), D7);
        U9 :  reg16 port map(reset, cd8, alu_out(15 downto 0), D8);
        U10 : reg16 port map(reset, cd9, alu_out(15 downto 0), D9);
        U11 : reg16 port map(reset, cd10, alu_out(15 downto 0), D10);
        U12 : reg16 port map(reset, cd11, alu_out(15 downto 0), D11);
        U13 : reg16 port map(reset, cd12, alu_out(15 downto 0), D12);
        U14 : reg16 port map(reset, cd13, alu_out(15 downto 0), D13);
        U15 : reg16 port map(reset, cd14, alu_out(15 downto 0), D14);
        U16 : reg16 port map(reset, cd15, alu_out(15 downto 0), D15);
        U17 : reg16 port map(reset, cd16, alu_out(15 downto 0), D16);
        U18 : reg16 port map(reset, cd17, alu_out(15 downto 0), D17);
        U19 : reg16 port map(reset, cd18, alu_out(15 downto 0), D18);
        U20 : reg16 port map(reset, cd19, alu_out(15 downto 0), D19);
        U21 : reg16 port map(reset, cd20, alu_out(15 downto 0), D20);
        U22 : compteur port map(reset, csp, decsp, ldsp, alu_out(15 downto 0), sp);
        U23 : compteur port map(reset, cptf, decptf, ldptf, alu_out(15 downto 0), ptf);
        U31 : drapeau port map(cdscr, load_reg, reset, set_i, reset_i, alu_out, dra(31 downto 28),
                               alu_out(4 downto 0), n1, c1, z1, i1, irq1);
        n <= n1;
        c <= c1;
        z <= z1; i <= i1;
        irq <= irq1;
        scr(4 downto 0) <= i1 & irq1 & z1 & c1 & n1;
        ir2 <= dra;

end comp;
```

### 3.1.4.1 La roue de fenêtres

La roue de fenêtres est constituée de 8 fenêtres contenant chacune 7 registres synchronisés sur un front montant, de 7 multiplexeurs ( «paramux», «mux8» et «muxout»), d'un circuit d'incrémentation de 16 bits, d'un compteur de 3 bits pouvant s'incrémenter ou se décrémenter et de plusieurs multiplexeurs tel que «clkfen». Les fenêtres sont numérotées de 0 à 7, des multiplexeurs sont utilisés pour sélectionner les sorties de la roue de fenêtres.

Tel que présenté au tableau 3.7, la roue de fenêtres a 11 entrées ( «reset», «sel», «jsr», «clk», «inc_pc», «ld», «selection», «bloc», «data_in», «prtf» et «prt») ainsi que 7 sorties ( «f0» à «f5» et «pc»). Lorsque «reset» est à 1, tous les registres sont mis à 0. Si «reset» et «sel» sont à 0, alors la valeur mise sur le bus de données («data_in») est transférée dans le registre de destination sélectionné par «prtf» et «prt». Pour effectuer cette opération, le démultiplexeur «clkfen» transfère le signal dans la fenêtre sélectionnée par «prtf» et ce signal est une fois de plus dirigé vers un démultiplexeur interne à la fenêtre qui permet le transfert de la donnée vers le registre choisi par le code de sélection «prt».

Afin de minimiser la surface de la roue, un seul additionneur est utilisé pour incrémenter les 8 compteurs de programmes PC. La sortie du compteur de la fenêtre courante est mise à l'entrée de l'additionneur. Si «ld» est à 0 et si on a une transition montante sur «inc_pc» alors le contenu du PC est incrémenté de 1. Dans le cas où le signal «ld» est à 1 et si on a une transition montante sur «inc_pc», le contenu du bus de données ( «Data_in») est transféré dans le PC de la fenêtre courante. Une telle approche nécessite l'usage d'un seul additionneur et d'un multiplexeur et elle minimise ainsi la surface de la roue de fenêtre.

Si les signaux «sel» et «jsr» sont à 1, ils indiquent à l'unité roue l'exécution de l'instruction JSR donc ils informent la roue de la possibilité de passer des paramètres. Alors, en mettant le contenu de la procédure appelante sur les bus de données «J0» et «J1», le processeur peut effectuer le passage de paramètres en un seul cycle d'horloge.

Les valeurs de «J0» et «J1» sont obtenues en utilisant le contenu de «prtf» décrémenté de 1 comme code de sélection des multiplexeurs «paramux» et elles sont mises aux entrées des 8 registres $F_0$ et $F_1$ respectivement. En utilisant, le démultiplexeur «clkfen» ainsi que le contenu de «sélection», le processeur peut passer les valeurs des registres contenues dans «J0» et «J1» comme arguments à la procédure appelée. «J0» est passé comme paramètre si «sélection(0)» est à 1 et «J1», si «sélection(1)» est à 1. Durant cette opération, la fenêtre de la procédure appelante est mise à la sortie de la roue.

Si le signal «sel» est à 1 et si le signal «jsr» est à 0, ils indiquent à la roue qu'il s'agit de l'exécution de l'instruction RTS et informent la roue de la possibilité de passer des paramètres. Alors, la roue utilise les mêmes principes précités, sauf que le code de sélection des multiplexeurs «paramux» est le contenu de «prtf» incrémenté de 1.

**TABLEAU 3.7**
**Modèle VHDL de l'unité «roue»**

**Unité «roue»**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;
entity roue is
        port(Reset, Clk, Sel, Jsr, Ld, inc_pc, bloc : in qsim_state;
        prt, prtf : in qsim_state_vector(2 downto 0);
        selection : in qsim_state_vector(1 downto 0);
        Data_in : in qsim_state_vector(15 downto 0);
        pc : out qsim_state_vector(15 downto 0);
        f5, f0, f1, f4, f3, f2 : out qsim_state_resolved_x_vector(15 downto 0));
end roue ;
architecture comp of roue is

component fen port
                        (reset, clk, inc_pc, sel, jsr : in qsim_state;
                         prt : in qsim_state_vector(2 downto 0);
                         selection : in qsim_state_vector(1 downto 0);
                         D0, D1 : in qsim_state_resolved_x_vector(15 downto 0);
                         Data_in, Data : in qsim_state_vector(15 downto 0);
                          f0, f1, f2, f3, f4, f5, pc : out qsim_state_vector(15 downto 0));
end component;
component clkfen port
                        (prtf : in qsim_state_vector(2 downto 0);
                         clk, inc_pc : in qsim_state;
```

**Unité «roue» (suite)**

```
                                clk0, clk1, clk2, clk3, clk4, clk5, clk6 : out qsim_state;
                                clk7, c0, c1, c2, c3, c4, c5, c6, c7 : out qsim_state));
end component;
component paramux port
                (sel : in qsim_state;
                prtf : in qsim_state_vector(2 downto 0);
                A0, A1, A2, A3, A4, A5, A6, A7, A8 : in qsim_state_vector(15 downto 0);
                J0 : out qsim_state_resolved_x_vector(15 downto 0) BUS);
end component;
component muxout port
                (prtf : in qsim_state_vector(2 downto 0);
                A0, A1, A2, A3, A4, A5, A6, A7 : out qsim_state_vector(15 downto 0);
                J0 : out qsim_state_resolved_x_vector(15 downto 0) BUS);
end component;
component mux8 port
                (prtf : in qsim_state_vector(2 downto 0);
                A0, A1, A2, A3, A4, A5, A6, A7 : out qsim_state_vector(15 downto 0);
                Z : out qsim_state_vector(15 downto 0));
end component;
signal f00,f01,f02,f03,f04,f05,f10,f11,f12,f13,f14, f15, f20, f21, f22, f75 : qsim_state_vector(15 downto 0);
signal f23,f24,f25,f30,f31,f32,f33,f34,f35,f40,f41,f42, f43, f44, f45, f74 : qsim_state_vector(15 downto 0);
signal f50, f51, f52, f53, f54, f55, f60, f61, f62, f63, f64, f65, f70, f71, f72, f73 : qsim_state_vector(15 downto 0);
signal pc0, pc1, pc2, pc3, pc4, pc5, pc6, pc7, pctampon, data, plus : qsim_state_vector(15 downto 0);
signal J0, J1 : qsim_state_resolved_x_vector(15 downto 0);
signal inc0,inc1,inc2,inc3,inc4,inc5,inc6,inc7,clk0,clk1,clk2,clk3,clk4,clk5,clk6,clk7 : qsim_state :='0';
signal s : qsim_state_vector(2 downto 0);
for u0, u1, u2, u3, u4, u5, u6, u7 : fen use entity WORK.fen(comp);
for u8, u9 : paramux use entity WORK.paramux(comp);
for u10 : mux8 use entity WORK.mux8(comp);
for u11 : clkfen use entity WORK.clkfen(comp);
for u14, u15, u16, u17 : muxout use entity WORK.muxout(comp);
begin
  u0 : fen port map(reset,clk0,inc0,sel,jsr,prt, selection, J0, J1, data_in, data, f00, f01, f02, f03, f04,
                    f05, pc0);
  u1 : fen port map(reset,clk1,inc1,sel,jsr,prt, selection, J0, J1, data_in, data, f10, f11, f12, f13, f14,
                    f15, pc1);
  u2 : fen port map(reset, clk2,inc2,sel,jsr,prt,selection, J0, J1, data_in, data, f20, f21, f22, f23, f24,
                     f25, pc2);
  u3 : fen port map(reset, clk3,inc3,sel,jsr,prt,selection, J0, J1, data_in, data, f30, f31, f32, f33, f34,
                    f35, pc3);
  u4 : fen port map(reset, clk4,inc4,sel,jsr,prt,selection, J0, J1, data_in, data, f40, f41, f42, f43, f44,
                    f45, pc4);
  u5 : fen port map(reset, clk5,inc5,sel,jsr,prt,selection, J0, J1, data_in, data, f50, f51, f52, f53, f54,
                    f55, pc5);
  u6 : fen port map(reset,clk6,inc6,sel,jsr,prt, selection, J0, J1, data_in, data, f60, f61, f62, f63, f64,
                    f65, pc6);
  u7 : fen port map(reset,clk7,inc7,sel,jsr,prt, selection, J0, J1, data_in, data, f70, f71, f72, f73, f74,
                    f75, pc7);
    process(jsr, sel, prtf)
    begin
if ((sel='0') or (bloc='1')) then s<=prtf; else if (jsr='1') then s<=prtf- "01"; else s<= prtf+"01"; end if;
    end process;
    u8 : paramux port map(sel, s, f00, f10, f20, f30, f40, f50, f60, f70, data_in, J0);
    u9 : paramux port map(sel, s, f01, f11, f21, f31, f41, f51, f61, f71, data_in, J1);
```

**Unité «roue» (suite)**

```
    u10 : mux8 port map(prtf, pc0, pc1, pc2, pc3, pc4, pc5, pc6, pc7, pctampon);
    u11 : clkfen port map(prtf, clk, inc_pc, clk0, clk1, clk2, clk3, clk4, clk5, clk6, clk7, inc0,
                          inc1, inc2, inc3, inc4, inc5, inc6, inc7);
    u14 : muxout port map(s, f02, f12, f22, f32, f42, f52, f62, f72, f2);
    u15 : muxout port map(s, f03, f13, f23, f33, f43, f53, f63, f73, f3);
    u16 : muxout port map(s, f04, f14, f24, f34, f44, f54, f64, f74, f4);
    u17 : muxout port map(s, f05, f15, f25, f35, f45, f55, f65, f75, f5);
          plus <= pctampon + "01";
          pc <= pctampon;
          f0 <= J0;
          f1 <= J1;
    process(ld, data_in, plus)
     begin
          if (ld='1') then data <= data_in; else data <= plus; end if;
    end process;
end comp;
```

### 3.1.5 L'unité de contrôle

Pour réaliser les diverses manipulations de contrôle et de séquencements lors de l'exécution d'une instruction, ou encore lors d'une réponse à une interruption, l'unité de contrôle est décomposée en trois sous-unités : l'unité «mécanisme», l'unité «interruption» et l'unité «décodeur». L'unité «mécanisme» permet de gérer les débordements de la pile intervenant lors de la sauvegarde ou du rétablissement du contenu des registres d'une fenêtre. L'unité «interruption» est utilisée pour contrôler les interruptions matérielles. Finalement, l'unité «décodeur» génère tous les signaux de contrôle utilisés lors de la lecture, du décodage et de l'exécution des instructions en réalisant les différents chronogrammes illustrés dans le deuxième chapitre aux figures 2.9 à 2.17. Les tableaux 3.8, 3.9 et 3.10 présentent le modèle VHDL de ces trois sous-unités.

**TABLEAUX 3.8**
**Modèle VHDL de l'unité «mécanisme»**

**Unité mécanisme**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;

entity mecanisme is
```

**Unité mécanisme (suite)**

```
        port( reset, jsr, clk, inc_fen, dec, set_pipe : in qsim_state;
              sel, store_in, load_reg_in, mem_rd_in : in qsim_state;
              ptf : in qsim_state_vector(15 downto 0);
              selm : out qsim_state_vector(4 downto 0);
              store, bloc, inc_sp, deco, load_reg, mem_rd, load_pc : out qsim_state;
end component;

architecture comp of mecanisme is
component horloge4 port
        (reset, clkin, jsr, set_pipe, bloc, sel : in qsim_state;
         clk, clk1,, clk2, clk3, clk4 : out qsim_state);
end component;
component bloc_horloge port
        (reset, reset_bloc, dec, inc_fen : in qsim_state;
         ptf : in qsim_state_vector(15 downto 0);
         bloc : out qsim_state);
end component;
signal rst_bloc, blocout, rst, clk0, clk1, clk2, dec0, clk3, clk4 : qsim_state :='0';
for U0 : bloc_horloge use entity WORK.bloc_horloge(comp);
for U1 : horloge4 use entity WORK.horloge4(comp);
begin
        bloc <= blocout;
        store <= store_in or (jsr and clk0 and not clk4 and not (clk3 and clk2 and clk1) and blocout;
        inc_sp <= ((not clk0 and not clk4 and jsr and (clk1 or clk2 or clk3)) or
                        (not clk0 and not jsr and clk4 and not (clk1 and clk2 and clk3))) and blocout;
        dec0 <= dec or (blocout and not jsr and clk4);
        deco <= dec0; rst <= reset or inc_fen;
        load_reg <= load_reg_in or (dec0 and clk0 and not (clk1 and clk2 and clk3);
        mem_rd <= mem_rd_in xor blocout;
        selm <= "11" & clk3 & clk2 & clk1;
    reset_bloc<=blocout and clk0 and ((jsr and clk4) and not clk1) or (not jsr and not clk4 and clk1));
        U0 : bloc_horloge port map(reset, rst_bloc, dec, inc_fen, ptf, blocout);
        U1 : horloge4 port map(rst, clk, jsr, set_pipe, blocout, sel, clk0, clk1, clk2, clk3, clk4);
end comp;
```

**TABLEAU 3.9**

**Modèle VHDL de l'unité «interruption»**

**Unité «interruption»**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;

entity interruption is port
        (IN_irq, IN_mni, IRQ, I : in qsim_state;
         fetch, clk1, clk, reset, swi1 : in qsim_state;
         swivec : in qsim_state_vector(2 downto 0);
         Data_in : in qsim_state_vector(15 downto 0);
```

**Unité «interruption» (suite)**

```
        int : out qsim_state;
        data_ir : out qsim_state_vector( 15 downto 0)) ;
end interruption;
architecture comp of interruption is
component sr port
        (Reset, s, r : in qsim_state;
        q : out qsim_state);
end component;
signal s1, s2, r1, r2, INMI, IIRQ : qsim_state :='0';
signal ir0, ir1, ir2, ir3, ir4 : qsim_state_vector(15 downto 0) :="0000000000000000";
for all : sr use entity WORK.sr(comp);
begin
        s1 <= IN_mni and not fetch and not clk1 and not clk;
        r1 <= INMI and fetch and clk1 and not clk;
        s2 <= IN_IRQ and IRQ and not I and not fetch and not clk1 and not clk;
        r2 <= IIRQ and fetch and clk1 and not clk;
        ir0 <= "1010100000000100";
        ir1 <= "0000000000000000";
        ir2 <= "1010100000000010";
        ir3 <= "0000000000000000";
        ir4 <= "000000000001" & swivec & '0';
        u0 : sr port map(reset, s1, r1, INMI);
        u1 : sr port map(reset, s2, r2, IIRQ);
        int <= INMI or IIRQ;
        PROCESS( INMI, fetch, clk1, IIRQ, swi1, ir0, ir1, ir2, ir3, ir4)
        begin
                if ((INMI='1') and (fetch='0') and (clk1='1')) then data_ir <= ir0;
                  elsif ((INMI='1') and (fetch='1') and (clk1='0')) then data_ir<= ir1;
                    elsif ((IIRQ='1') and (fetch='0') and (clk1='1')) then data_ir<= ir2;
                      elsif ((IIRQ='1') and (fetch='1') and (clk1='0')) then data_ir <= ir3;
                        elsif ((swi1='1') and (fetch='1') and (clk1='0')) then data_ir <= ir4;
                            else data_ir <= data_in;
                end if;
        end process;
end comp
```

## TABLEAU 3.10

## Modèle VHDL de l'unité «décodeur»

**Unité «decodeur»**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;
entity decodeur is port
        (ir2 : in qsim_state_vector(31 downto 27);
         ir1 : in qsim_state_vector(31 downto 28);
         mode : in qsim_state_vector(20 downto 16);
         fetch, clk1, clk, n, c, z, int : in qsim_state;
```

**Unité «decodeur» (suite)**

```
        mem_rd, load_ptr, load_irh, load_reg, load_irl, inc_pc, set_pipe, ind, dir : out qsim_state;
        inc_fen, reset_i, load_pc, dec, sel, jsr, store, set_i, pb : out qsim_state);
end decodeur;

architecture comp of decodeur is

signal alu, alu2, mem_rd1, inc_pc1, ind1, dir1 : qsim_state :='0';
signal je, jm, jc, inc_pc2, sel0, mem_rd2, jmp1 : qsim_state :='0';
begin
        mem_rd1 <= (fetch xor clk1) and not int;
        mem_rd2 <= (dir1 or ind1) and not fetch and not clk1;
        mem_rd <= mem_rd1 or mem_rd2;
        inc_pc1 <=(fetch and clk) and not int;
        inc_pc <= inc_pc1 or inc_pc2;
        load_irh <= not fetch and clk1 and not clk;
        load_irl <= fetch and not clk1 and not clk;
        set_pipe <= fetch and clk1 and not clk;

        process(ir2(31 downto 27))
        begin
                if (ir2(31 downto 29) :="001") or (ir2(31 downto 29) :="010")) then alu<='1';
                  elsif (ir2(31 downto 27)="00001" then alu <='1';
                    elsif (ir2(31 downto 27)="00010") then alu <='1';
                      else alu <='0';
                end if;
        end process;

        load_reg <= (alu and fetch and not clk1 and not clk) or (alu2 and not fetch and not clk1 and not clk);
        load_ptr <= (alu and fetch and not clk1) or (alu2 and not fetch and not clk1 and not sel0);
        ind1 <= mode(18) and not mode(17) and mode(16);
        dir1 <= not mode(18) and not mode(17) and mode(16);
        ind <= ind1; dir <= dir1;
        process(ir2)
        begin
                if ((ir2="10000") or (ir2(31 downto 28)="1010") then jmp1 <='1'; else jmp1 <='0';
                end if;
                if (ir2="10001") then je <='1'; else je <='0'; end if;
                if (ir2="10010") then jm <='1'; else jm <=`0`; end if;
                if (ir2="10011") then jc <='1'; else jc <='0'; end if;
        end process;
        inc_pc2 <= (jmp1 or (mode(20) and z) or (je and (z xor (mode(19))) or (jm and (n xor mode(19))) or
                    (jc and (c xor mode(19)))) and (not fetch and not clk1 and not clk);
        load_pc <= (jmp1 or je or jc or jm) and not fetch and not clk1;
        inc_fen <= ir1(31) and not ir1(30) and ir1(29) and fetch and clk1 and not clk;
        dec <= ir1(31) and not ir1(29) and ir1(29) and ir1(28) and fetch and clk1;
        sel <= sel0; sel0 <= ir2(31) and not ir2(30) and ir2(29) and not fetch and not clk1;
        jsr <= ir2(31) and not ir2(30) and ir2(29) and ir2(28) and not ir2(27) and not fetch and not clk1;
        process(ir2, fetch, clk1, clk)
        begin
                if ((ir2="10101") or (ir2="11010")) then set_i<=not fetch and not clk1 and not clk;
                        else set_i<='0';
```

**Unité «decodeur» (sutie)**

```
                end if;
        end process;
        process(ir2, fetch, clk1, clk)
        begin
                if ((ir2="10111") or (ir2="11011")) then reset_i <= fetch and clk1 and not clk;
                        else reset_i <='0';
                end if;
        end process;
        process(ir2, fetch, clk1, clk)
        begin
            if (ir2="11001") then store <= not fetch and not clk1 and not clk; else store <='0'; end if;
        end process;
        process(mode(18 downto 17), ir2)
        begin
                if ((ir2(31 downto 30)="10") and (mode(18 downto 17)="00") then pb <='1';
                   elsif ((ir2="10101") or (ir2="11001")) then pb <='1';
                    elsif ((ir2(31 downto 28)="1011") or (ir2(31 downto 28)="1100")) then pb <='1';
                      else pb <='0';
              end if;
        end process;
end comp;
```

### 3.1.6 L'unité arithmétique et logique (UAL)

L'UAL est un circuit combinatoire pouvant effectuer des opérations arithmétiques et logiques entre les entrées «A» et «B». La sortie de l'UAL contient le résultat de l'opération sélectionnée par l'entrée «IR», ou le contenu de l'entrée «B» (lorsque «passb» est à un niveau logique haut), ou encore l'entrée «Data_in» (lorsque «mem» est au niveau logique 1). Le modèle VHDL de l'UAL est représenté au tableau 3.11.

**TABLEAU 3.11**
**Modèle VHDL de l'UAL**

**Unité ALU**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;
entity alu is
          port( IR : in qsim_state_vector(31 downto 27);
                A, B : in qsim_state_vector ( 16 downto 0) ;
                data_in : in qsim_state_vector(16 downto 0);
                 cin, passb, mem      : in qsim_state;
                SORTIE : out qsim_state_vector(16 downto 0));
end alu ;
architecture comp of alu is
signal M1, M2 : qsim_state_vector( 16 downto 0) :="00000000000000000";
```

**Unité ALU (suite)**

```
begin
        process(A, B)
        begin
                if ( A > B) then M2 <= A; M1 <= B; else M2 <= B; M1 <= A; end if;
        end process;
        process(passb, cin, IR, a, b, m1, m2, mem, data_in)
        begin
                if (mem='1') then sortie <= data_in;
                  elsif (passb ='1') then sortie <= B;
                    elsif IR="00001") then Sortie <=A+B;
                     elsif (IR(31 downto 28) ="0001") then Sortie <= A-B;
                      elsif (IR ="00100") then Sortie <= A xor B;
                       elsif (IR ="00101") then Sortie <= A or B;
                        elsif (IR ="00110") then Sortie <= A and B;
                          elsif (IR ="00111") then Sortie <= '0' & B(16 downto 1);
                           elsif (IR ="01000") then Sortie <= B(0) & cin & B(15 Downto 1);
                            elsif (IR ="01001") then Sortie <= B(15 downto 0) & cin;
                             elsif (IR ="01010") then Sortie <= M1;
                              else Sortie <= M2;
                end if;
        end process;
end comp;
```

## 3.2 La synthèse et l'optimisation

Après que toutes les sous-unités ont été simulées et assemblées, il faut effectuer la synthèse de la description VHDL du processeur en un ensemble de portes logiques. Ensuite, le circuit obtenu est optimisé avec des contraintes de temps et de surface. Ces opérations ont été effectuées avec le logiciel Autologic de Mentor Graphics et la librairie des circuits FPGAs de la famille XILINX-4000. Le résultat obtenu, c'est-à-dire le circuit final, est transformé en un format compatible avec les outils logiciels de l'environnement XACT de XILINX qui sont utilisés, par après, pour la réalisation du processeur.

## 3.3 La réalisation

La réalisation de processeurs peut se faire avec plusieurs types de technologies de circuits tels que : les composants normalisés ou programmables, les prédiffusés, les cellules normalisées et les circuits dédiés. Généralement, les concepteurs de circuits intégrés cherchent à obtenir une performance optimale et à réduire le temps de conception, le temps de fabrication ainsi que le coût de production. Les composants

programmables représentent une solution avantageuse qui respecte, pour un faible taux de production, les différents critères précités.

Le processeur a été conçu pour être mis en oeuvre avec des circuits FPGAs ( «*Field Programmable Gate Arrays*») reprogrammables de la famille XILINX-4000. Ces circuits réutilisables, performants, reconfigurables et économiques permettent d'obtenir très rapidement un prototype fonctionnel.

Les circuits FPGAs de XILINX sont regroupés en trois familles : XILINX-2000, XILINX-3000 et XILINX-4000. La complexité et le nombre de cellules disponibles représentent la principale différence à l'intérieur d'une famille ou entre les différentes familles. Le tableau 3.12 illustre quelques caractéristiques des circuits de la famille XILINX-4000.

Ces derniers sont organisés en matrice de cellules, chacune d'elles, peuvant être reliée par des connexions locales et globales programmables. Ces circuits contiennent essentiellement trois types de ressources : les blocs d'entrées et de sorties (IOB), les cellules logiques (CLB) et les interconnexions. Les cellules logiques (CLB) sont regroupées sous forme de matrice de cellules constituées d'éléments logiques programmables. Par exemple, dans le cas du circuit XC4013, sa structure interne repose sur une matrice de 24x24 CLBs contenant chacune 13 entrées, 4 sorties, des éléments logiques universels et des registres.

**TABLEAU 3.12**

**Caractéristiques des circuits FPGAs de la famille XILINX-4000**

| **Circuits** | Portes logiques éq. | Matrice de CLBs | Nombre de bascules | Décodeur d'entrées | N. d'entrées et de sorties | Nombre de CLBs |
|---|---|---|---|---|---|---|
| **XC4003** | 3000 | 10x10 | 360 | 30 | 80 | 100 |
| **XC4005** | 5000 | 14x14 | 616 | 42 | 112 | 196 |
| **XC4006** | 6000 | 16x16 | 768 | 48 | 128 | 256 |
| **XC4008** | 8000 | 18x18 | 936 | 54 | 144 | 324 |
| **XC4010** | 10 000 | 20x20 | 1 120 | 60 | 160 | 400 |
| **XC4013** | 13 000 | 24x24 | 1 536 | 72 | 192 | 576 |
| **XC4025** | 25 000 | 32x32 | 2 560 | 96 | 256 | 1 024 |

Telle que représentée à la figure 3.5, un CLB dispose de deux entrées de 4 bits ($F_1$ à $F_4$ et G1 à $G_4$) reliées à deux générateurs de fonctions logiques pouvant réaliser n'importe quelle fonction avec une entrée de 4 bits ou moins. La table de vérité de la fonction logique désirée est programmée dans des éléments de mémorisation. Un troisième générateur de fonctions logiques (H) à trois entrées permet de combiner le résultat des deux autres générateurs avec une entrée externe à la cellule logique (H1). L'unité S/R permet de déterminer les caractéristiques de mise à 0 ou à 1 des deux registres de la cellule.

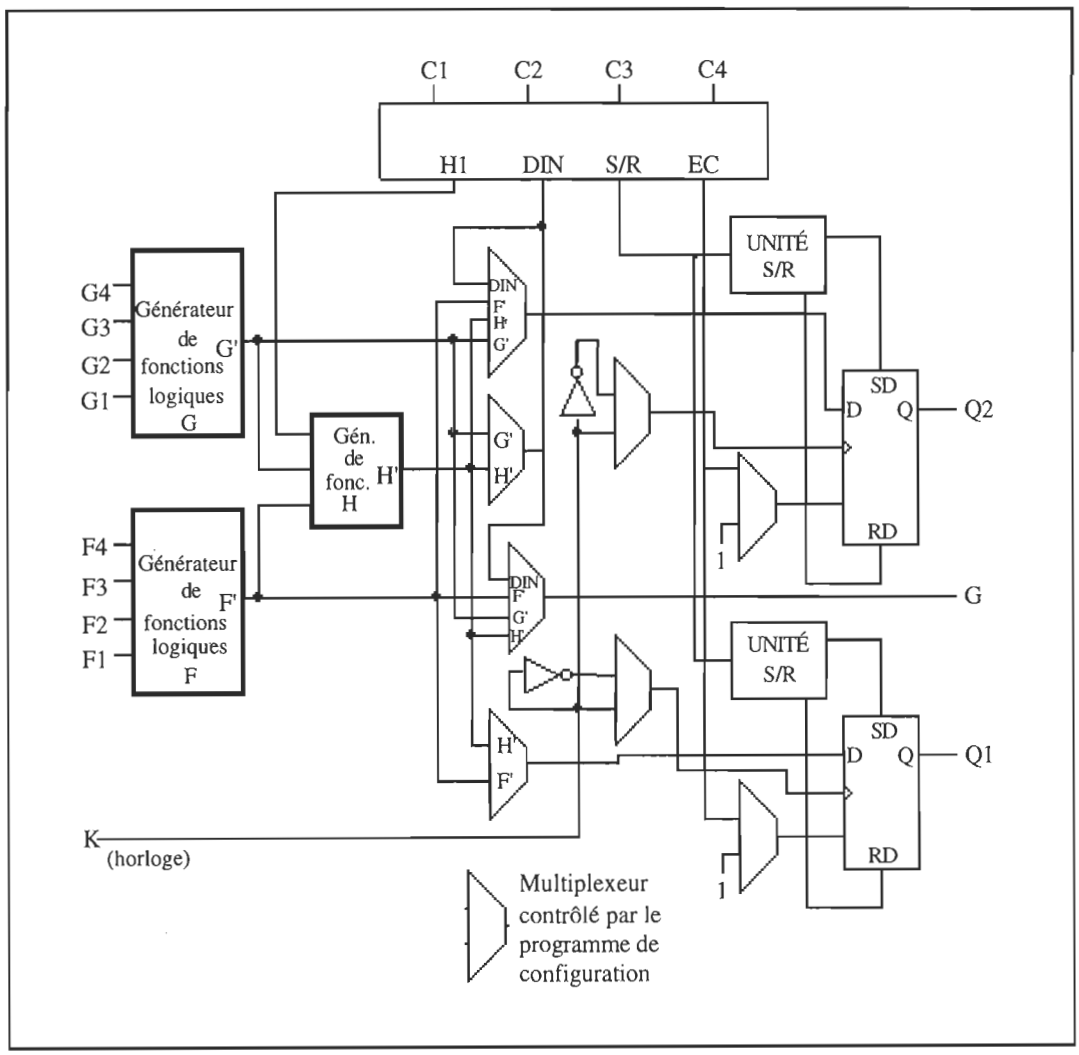


**Figure 3.5. Diagramme bloc des cellules logiques (CLB)**

Les cellules d'entrées et de sorties (IOB) permettent de relier la logique interne d'un circuit avec les broches externes. La figure 3.6 illustre une IOB qui, en sortie, permet d'inverser, au besoin, le signal (OUT) connecté à la broche externe. Cette connexion peut être directe, mémorisée dans une bascule ou encore contrôlée par un tampon à trois états. Ce dernier permet d'obtenir une configuration bidirectionnelle de la broche en utilisant le signal de validation de la sortie OE. Chaque IOB peut fournir un courant de 12 mA avec un temps de transition optimisé. En entrée, la broche est connectée à $I_1$ et à $I_2$ avec une liaison directe ou mémorisée dans une bascule tout en permettant de réduire le délai à l'entrée de cette bascule.

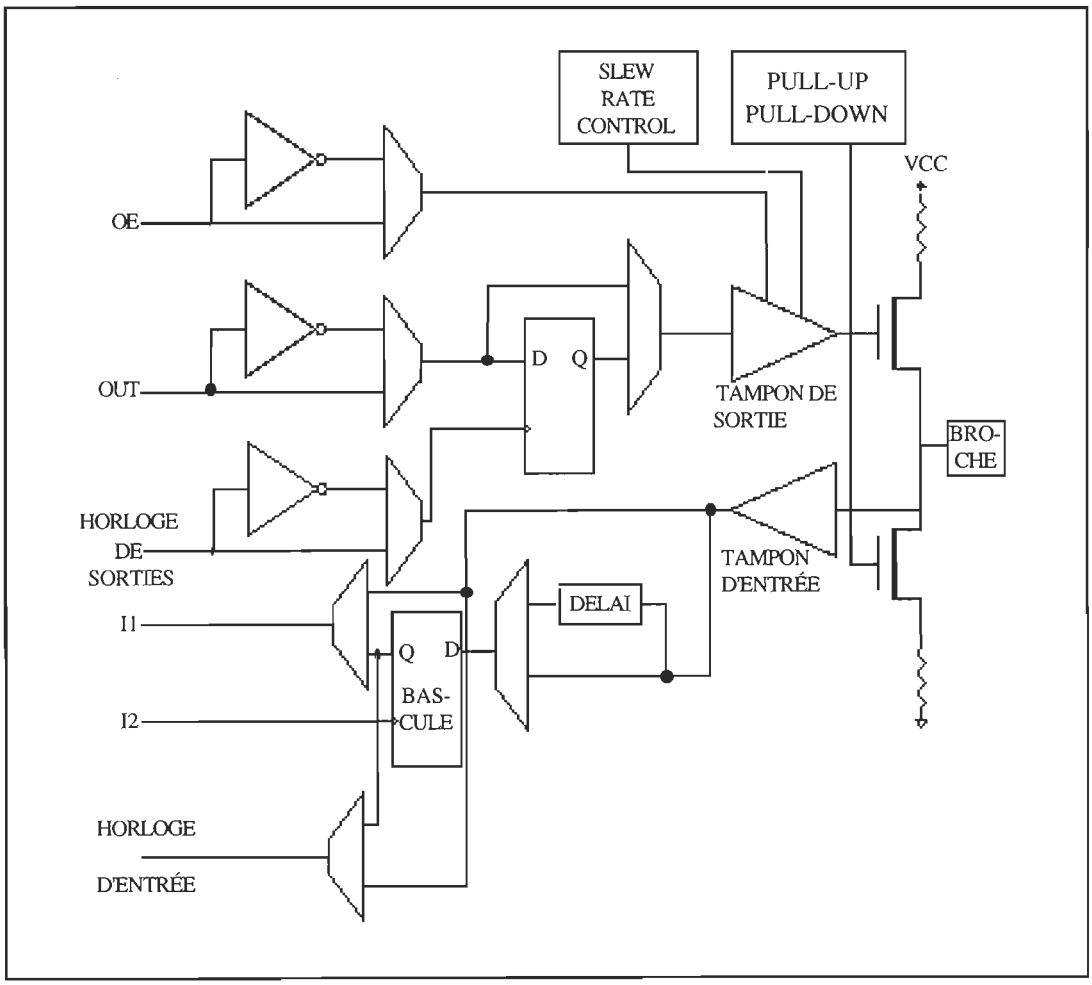


**Figure 3.6. Diagramme bloc des cellules d'entrées et de sorties (IOB)**

Le réseau d'interconnexions est constitué par plusieurs segments de métal permettant de réaliser un routage quelconque avec une résistance et une capacitance moyenne minimale. Principalement, il est formé par trois types de lignes : les lignes simples, les lignes doubles et les lignes longues.

Les lignes simples sont constituées par des bandes verticales et horizontales ayant à leur intersection des matrices de commutation qui, par programmation, permettent d'établir de nombreuses connexions locales.

Les lignes doubles, regroupées par paire et sur des bandes alternées, sont constituées par des segments de métal qui traversent deux CLBs avant d'atteindre une matrice de commutation et elles ont deux fois la longueur des lignes simples. Toutes les entrées et toutes les sorties des CLBs (sauf l'horloge K) peuvent être reliées aux lignes doubles adjacentes.

Les lignes longues sont constituées par des segments de métal traversant la matrice de CLBs d'un côté à l'autre. Chaque ligne dispose, dans son centre, d'un interrupteur programmable qui la divise en deux parties égales et indépendantes tout en permettant de connecter toutes les entrées et les sorties des CLBs avec celle-ci. Les sorties des CLBs peuvent aussi utiliser une liaison avec des lignes simples ou encore avec des tampons à trois états. Les connexions entre les lignes longues et simples sont gérées par des points d'interconnexions programmables. Toutes les unités programmables sont constituées par des cellules de mémoires RAM statiques. Cette approche permet :

1- d'obtenir une configuration pouvant changer les fonctions logiques pendant l'exécution et d'obtenir ainsi une très grande flexibilité;
2- d'effectuer des modifications ou des mises à jour du circuit très rapidement;
3- de configurer dynamiquement le circuit pour réaliser les différentes fonctions à des temps différents;
4- de configurer le circuit pour différents environnements, d'obtenir un circuit à plusieurs fonctions ou encore de faciliter la correction d'erreurs;
5- de réduire le temps de conception.

Les outils logiciels de l'environnement XACT de XILINX permettent de traduire le résultat de la synthèse et de l'optimisation en un fichier pouvant configurer le circuit choisi. Le processeur a été réalisé avec deux circuits FPGAs de XILINX (XC4013PG223). Telle que présentée au tableau 3.13, le premier FPGA est utilisé exclusivement pour la réalisation de la «roue» de fenêtres, tandis que le deuxième est assigné au restant des circuits du processeur.

**TABLEAU 3.13**
**Ressources utilisées par le processeur**

| **Ressources (XC4013PG223)** | **Unité «roue»** | **Les autres unités** |
|---|---|---|
| IOB | 142/192 (74%) | 180/192 (94%) |
| CLB | 496/1152 (43%) | 1054/1152 (91%) |
| Bascule | 899/1152 (78%) | 442/1152 (38%) |
| Ressources de routage (bus) | 96/96 (100%) | 80/96 (83%) |

# CHAPITRE IV

## ARCHITECTURE DU PROCESSEUR FLOU

Tout système de contrôle fournit un ensemble de sorties selon un ensemble d'entrées en utilisant un algorithme quelconque. Par exemple, un processeur peut lire dans un tableau la valeur des sorties à produire pour chacune de ses entrées, ou pour chaque combinaison de celles-ci. Toutefois, pour un nombre significatif d'entrées et de sorties, cette approche devient très peu efficace car la taille du tableau, donc de la mémoire, augmente considérablement.

Afin de réduire la taille de la mémoire utilisée, les processeurs ordinaires exécutent un algorithme mathématique qui leur permet de modifier les sorties en fonction des entrées. Pour un procédé très complexe ou mal compris, le modèle mathématique peut être difficile à trouver ou inutilisable en temps réel, ou encore trop compliqué. Lorsqu'une très grande précision n'est pas requise et lorsqu'un humain peut gérer le processus, la logique floue peut être utilisée et un processeur spécialisé conçu pour la logique floue peut alors contrôler le procédé. L'architecture des processeurs flous permet d'obtenir des processeurs très rapides, faciles à concevoir et à adapter aux conditions de fonctionnement du processus. Cette architecture est simple à implanter et très flexible pour les utilisations particulières, elle requiert très peu de mémoire et elle permet de simuler certains comportements humains. Elle utilise des règles floues pour simuler le raisonnement humain, ce qui diffère des processeurs ordinaires qui utilisent des instructions pour exécuter un algorithme mathématique quelconque.

Le processeur flou comporte un mécanisme décisionnel en logique floue qui permet d'effectuer un contrôle basé sur les règles de cette logique et de prendre une décision

adéquate en temps réel. Telle que représentée à la figure 4.1, la réalisation du processeur flou consiste à concevoir un circuit capable de réaliser les trois opérations suivantes : la fuzzification, l'évaluation des règles et la défuzzification (Won, 1994).

## 4.1 La fuzzification

Le traitement de l'entrée est effectué par un processeur hôte qui transforme la valeur de cette entrée exacte en une valeur floue tout en tenant compte des erreurs pouvant être engendrées à cette étape. Cette valeur floue est obtenue en calculant (avec une certaine correction d'erreurs si nécessaire) la contribution de cette valeur exacte aux divers sous-ensembles flous représentant l'entrée. Une valeur floue est alors un vecteur contenant le degré d'appartenance $(\mu_t)$ des sous-ensembles flous pour une valeur exacte quelconque.

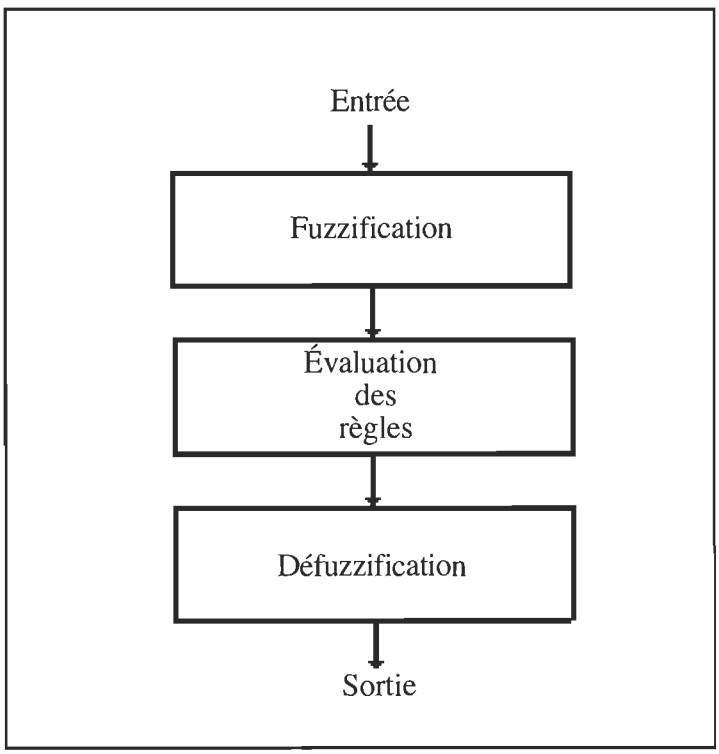


**Figure 4.1. Opérations d'un processeur flou**

Pour représenter un élément d'un vecteur flou, le processeur flou utilise 6 bits : $\mu_t$ = "00"représente aucune appartenance, $\mu_t$ = "3*F*", l'appartenance totale et les autres valeurs représentent des nombres intermédiaires dans l'intervalle [0..1]. Par exemple, dans la figure 4.2, la transformation de l'entrée exacte (78° F) produit une entrée floue *D = {0.58, 0.22}*. Une valeur floue (*D*) est mise dans un tableau disposé dans un banc de mémoire contenant 64 cases de 6 bits chacune. Le processeur flou peut alors accéder à cette valeur, c'est-à-dire les éléments de l'entrée floue, en consultant ce tableau. Cette méthode s'appelle la consultation de tableau. Pour réaliser la défuzzification, le processeur flou utilise trois registres de 6 bits : «N», «$N_j$» et «Fuzzy».

Le registre «N» contient la longueur du vecteur calculé par le processeur hôte et il indique ainsi le nombre de sous-ensembles $\left(t_i\right)$ représentant l'entrée exacte. Le processeur hôte, en mettant les entrées «sel» à 1 et «sel_r» à 0 et en envoyant une transition sur l'entrée «clk», peut écrire une valeur comprise entre 1 à 63 dans ce registre. Quant au registre «$N_j$», il est constitué d'un compteur synchrone et il sert de bus d'adresses pour le banc de mémoire contenant la valeur floue (*D*). Finalement, le registre «Fuzzy» permet de contenir un des éléments $\left(\mu_{t_i}(e)\right)$ du vecteur de l'entrée (*D*) situé dans le tableau. Les tableaux 4.1 et 4.2 présentent les modèles VHDL d'un registre et d'un compteur synchrone de 6 bits respectivement.

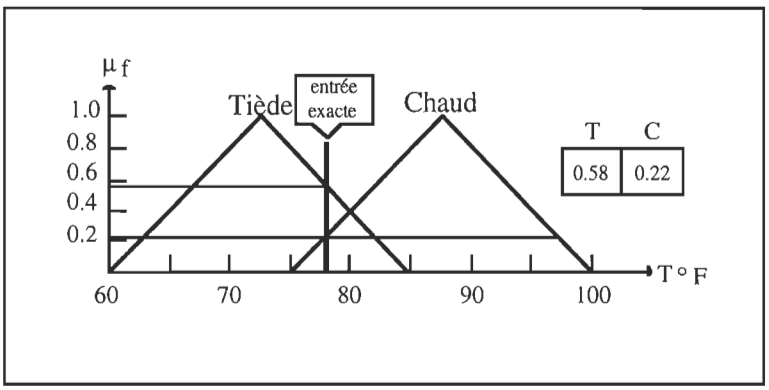


**Figure 4.2. Fuzzification**

## TABLEAU 4.1

## Modèle VHDL d'un registre de 6 bits

**Unité «reg6»**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;
entity Reg6 is port(Reset, Clk : in qsim_state;
                    Data_in : in qsim_state_vector(5 downto 0 );
                    Data_out : out qsim_state_vector(5 downto 0)
                    );
end Reg6;
architecture comp of Reg6 is
signal data_rst : qsim_state_vector(5 downto 0) :="000000";
begin
  Reg_6 : process(Clk, Reset, data_rst)
          begin
               if (Reset ='1') then Data_out <=  data_rst ;
                  elsif ( (Clk ='1') and (Clk'event) and (Clk'last_value ='0')) then
                        Data_out <= Data_in;
               end if;
          end process;
end comp;
```

## TABLEAU 4.2

## Modèle VHDL d'un compteur synchrone de 6 bits

**Unité compteur6**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;
entity compteur6 is port ( Reset, Clk, dec , Ld : in qsim_state;
                           Data_out : out qsim_state_vector(5 downto 0));
end compteur6 ;
architecture comp of compteur6 is
signal D0, D1 : qsim_state_vector(5 downto 0) := "000000";
component reg6 port (  Reset, Clk : in qsim_state;
                       Data_in : in qsim_state_vector(5 downto 0);
                        Data_out : out qsim_state_vector(5 downto 0));
end component;
for u0 : reg16 use entity WORK.reg6(comp);
begin
          D0 <= D1 + "01";
          u0 : reg6 port map ( Reset, Clk, D0, D1);
          Data_out <= D1;
end comp;
```

## 4.2 L'évaluation des règles

L'évaluation des règles floues permet de déterminer l'action à prendre en fonction de l'entrée floue et des règles floues placées dans un tableau situé dans un banc de mémoire réservée de 4 Kbits. Lors de cette opération, le processeur utilise la règle de composition «max-min» (voir section 1.4 et équation 1.20) pour calculer l'apport de la valeur floue au résultat final. Tel que représenté à la figure 4.3, le circuit réalisant la composition «max-min» est constitué de cinq sous-unités : de deux sous-unités «min», d'une sous-unité «max» et de deux registres.

La première sous-unité «min» permet de déterminer la valeur minimale entre un élément de l'entrée floue et celui des règles. La sous-unité «max» calcule la contribution maximale d'un élément de l'entrée floue sur un des sous-ensembles flous représentant la sortie et le résultat de celle-ci est conservé temporairement dans un registre. Une deuxième sous-unité «min» permet de traiter d'autres types de règles avec plusieurs entrées et sa sortie peut être enchaînée à un autre processeur flou. Par exemple, le processeur peut traiter les règles à deux entrées telles que «**Si** *observation 1* et *observation 2* **Alors** *action*». Une des deux entrées doit provenir d'un autre processeur. Finalement, le deuxième registre conserve l'action de chaque règle. Lors de l'évaluation des règles floues, le processeur mesure l'action de chaque règle et, ensuite, il la combine au résultat final afin d'obtenir à la défuzzification une sortie exacte.

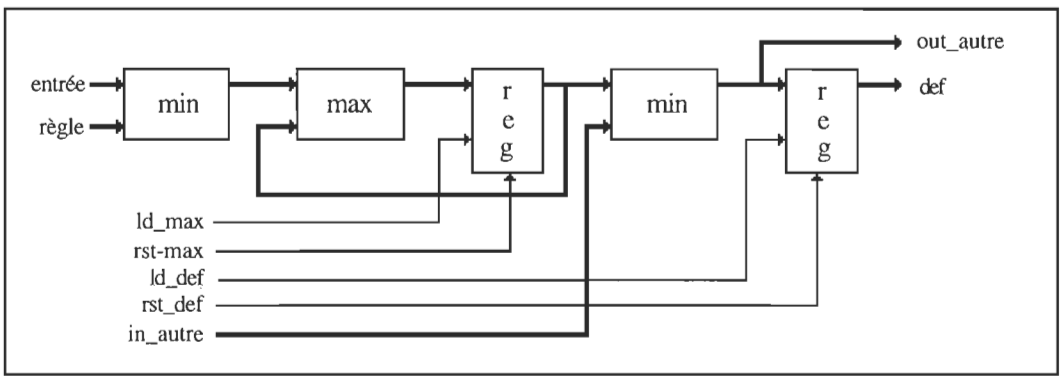


**Figure 4.3. Diagramme bloc du circuit réalisant la composition «max-min»**

Pour compléter cette opération, le processeur flou utilise également trois registres de 6 bits : «M», «$M_i$» et «Evaluation». Le registre «M» indique le nombre de sous-ensembles flous représentant la sortie. Pour y accéder, le processeur hôte met les entrées «sel» et «sel_r» à 1 et, ensuite, il envoie une transition sur «clk» pour entamer l'écriture de la donnée dans ce registre. Quant au registre «$M_i$», il est constitué d'un compteur synchrone et, combiné avec le registre «$N_i$», il sert de bus d'adresses pour le banc de mémoire contenant les différentes règles établies. Finalement, le registre «Evaluation» contient la valeur d'un élément du tableau représentant les règles floues. L'unité «si_alors» réalise la composition «max-min» et son modèle VHDL est représenté au tableau 4.3.

**TABLEAU 4.3**

**Modèle VHDL de l'unité «si_alors»**

**Unité «si-alors»**

```
LIBRARY MGC_PORTABLE;
use MGC_PORTABLE.QSIM_LOGIC.ALL;
use MGC_PORTABLE.QSIM_RELATIONS.ALL;
entity si_alors is
        port( ld_max, rst_max, ld_def, rst_def : in qsim_state;
              entree, regle, in_autre : in qsim_state_vector(5 downto 0);
              out_autre, def : out qsim_state_vector(5 downto 0));
end component;
architecture comp of si_alors is
component max port
        (a, b : in qsim_state_vector(5 downto 0); z : out qsim_state_vector(5 downto 0));
end component;
component min port
        (a, b : in qsim_state_vector(5 downto 0); z : out qsim_state_vector(5 downto 0));
end component;
component reg6 port
        ( reset, Clk : in qsim_state; data_in : in qsim_state_vector(5 downto 0 );
          data_out : out qsim_state_vector(5 downto 0));
end component;
signal z1, z2, z3, z4 : qsim_state_vector(5 downto 0) :="000000";
for U0, U3 : min use entity WORK.min (comp);
for U1 : max use entity WORK.max (comp);
for U2, U4 : reg6 use entity WORK.reg6 (comp);
begin
        U0 : min port map(entree, regle, z1);
        U1 : max port map(z1, z2, z3);
        U2 : reg6 port map(rst_max, ld_max, z3, z2);
        U3 : min port map(in_autre, z2, z4);
        U4 : reg6 port map(rst_def, ld_def, z4, def); out_autre <= z4;
end comp;
```

## 4.3 La défuzzification

Après avoir mesuré l'effet d'une règle, la défuzzification permet de calculer sa contribution au résultat final. Lors de cette opération, chacune des fonctions caractéristiques représentant la sortie est limitée par la valeur calculée à l'évaluation des règles. Ensuite, on calcule le centre de gravité de la fonction obtenue afin d'obtenir la sortie :

$$Sortie = \frac{\sum \mu(n) \bullet n}{\sum \mu(n)}. \qquad 4.3$$

Cette technique est appelée la méthode du centre de gravité (COG) et elle est illustrée à la figure 4.4.

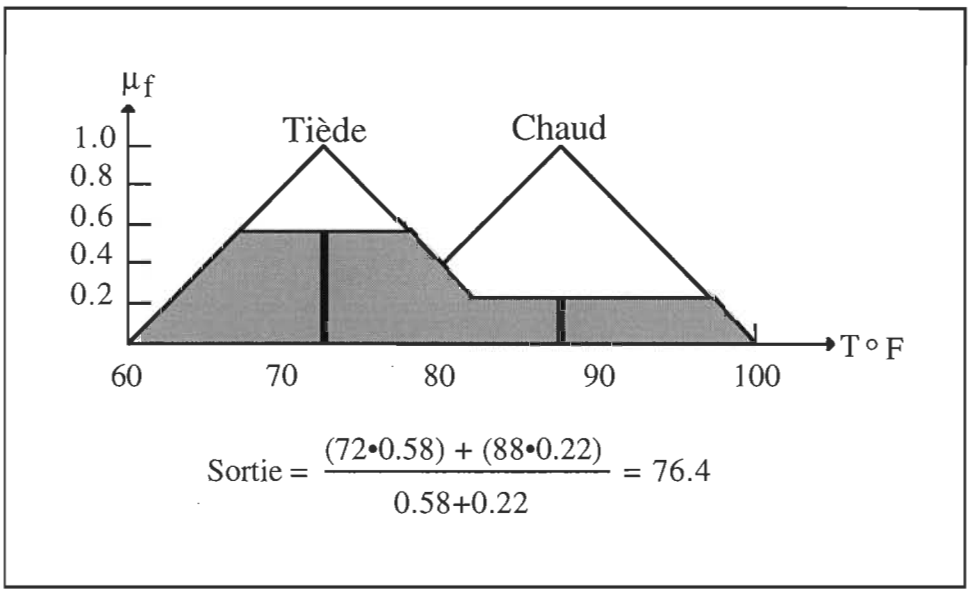


**Figure 4.4. Méthode du centre de gravité (COG)**

Les valeurs des degrés d'appartenance des sous-ensembles flous représentant la sortie ainsi que les différentes valeurs de «n» dans l'équation 4.3 sont arbitrairement mises dans deux tableaux placés dans deux bancs de mémoire adressés par le registre «$M_i$».

Le processeur peut, au besoin, les consulter pour effectuer les différents calculs nécessaires à la défuzzification. Tel que représenté à la figure 4.5, le circuit de défuzzification est composé de 7 unités : une unité «min», deux additionneurs, deux registres, un multiplicateur et un diviseur. L'unité «min» permet de limiter la contribution d'une règle floue. Les deux additionneurs permettent de calculer les sommes de l'équation 4.3 et leurs résultats sont conservés temporairement dans deux registres. Le multiplicateur effectue la multiplication de l'équation 4.3 et, finalement, le diviseur effectue la division des contenus de ces registres.

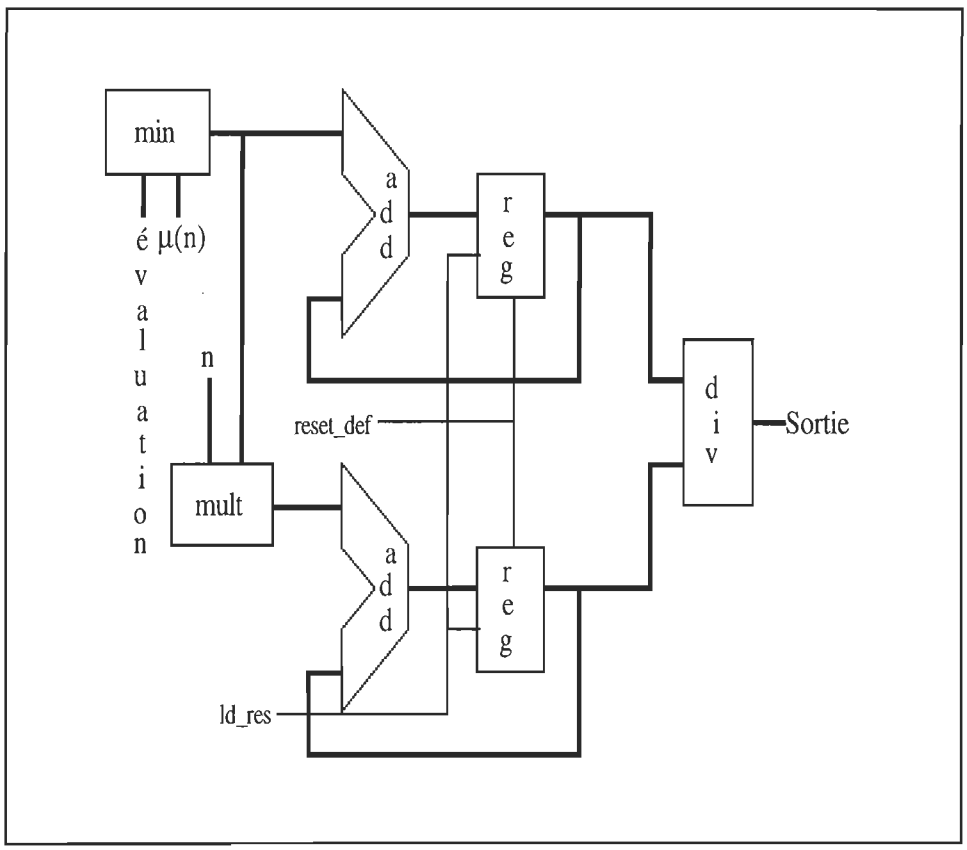


**Figure 4.5. Diagramme du circuit de défuzzification**

Cette architecture du processeur flou est complétée par deux autres unités : «horloge» et «decflou». L'unité «horloge» génère les différents signaux d'horloge nécessaires pour ordonner chronologiquement les différents signaux de contrôles utilisés lors de la fuzzification, lors de l'évaluation des règles et lors de la défuzzification. Quant à l'unité

«decflou», elle génère ces différents signaux de contrôles représentés à la figure 4.6, permettant ainsi de réaliser les différentes étapes d'un processeur flou. Les signaux «mem_rd», «ld_def» et «ld_fuz» permettent l'accès en parallèle à des données situées dans plusieurs bancs de mémoire. Le signal «ld_max» permet d'incrémenter le registre «$N_j$» tout en sauvegardant dans un registre le maximum entre deux éléments et, d'une manière séquentielle, il permet ainsi de réaliser la composition «max-min».

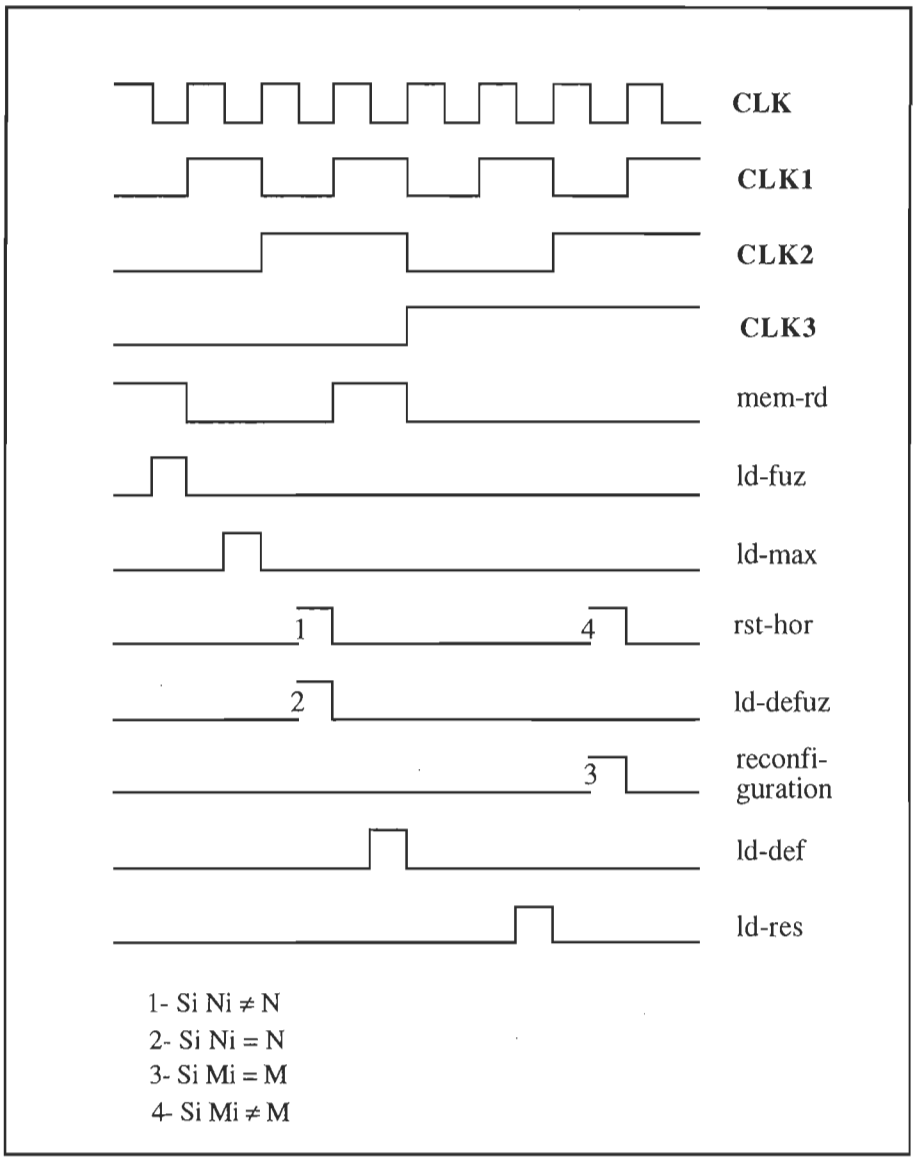


**Figure 4.6. Chronogramme du décodeur «decflou»**

Le signal «reconfiguration» permet d'activer le circuit de la division et le signal «rst_hor» effectue une remise à 0 de l'unité «horloge». Finalement, le signal «ld_res» permet d'incrémenter le registre «$M_i$», de faire une remise à 0 du registre après l'évaluation d'une règle et de conserver temporairement dans des registres les résultats de la défuzzification.

# CHAPITRE V

# CONCLUSION

Ce mémoire décrit l'architecture d'un nouveau processeur RISC de 16 bits spécialisé pour la logique floue. Le processeur est composé de 8 unités : l'UAL, 3 multiplexeurs, un banc de registres, un démultiplexeur, une unité d'horloge et une unité de contrôle (voir fig. 2.7). Le banc de registres est constitué de 81 registres dont 32 sont accessibles pour une instruction donnée : le registre de statut et de contrôle (SCR), la pile (SP), le pointeur de fenêtres (PTF), les 21 registres à usage général ($D_0$ à $D_{20}$), la roue de fenêtres (56 registres) et le registre d'instructions (IR). Ce dernier permet d'obtenir une structure d'exécution *pipelinée* à deux étages. Tous ces registres sont de 16 bits sauf les registres SCR (5 bits) et IR (32 bits).

La roue est formée de 8 fenêtres constituées chacune de 6 registres (F0 à F5) et un compteur de programme (PC). Elle permet, lors des appels et des retours de sous-routines, de conserver l'adresse de retour, d'effectuer le passage de deux paramètres au maximum et de contenir des variables locales. Pour les imbrications de sous-routines supérieures à huit, elle génère une exception qui permet de rétablir ou de sauvegarder les registres de la fenêtre courante. La structure de la roue est simple, nouvelle, originale et très performante. Elle permet de réduire le surcoût des appels et des retours de procédures.

L'unité arithmétique et logique (UAL) de 16 bits réalise les diverses opérations arithmétiques et logiques et les résultats de celle-ci modifient les drapeaux «N», «C» et «Z» du SCR. Finalement, les autres unités effectuent les différents contrôles de séquencement nécessaires à l'exécution d'une instruction. Un multiplexeur permet de sélectionner une donnée parmi plusieurs et de mettre celle-ci sur une des deux entrées de l'UAL ou sur le bus d'adresse. Le démultiplexeur permet d'envoyer un signal

d'horloge au registre sélectionné. L'unité «CLOCK» produit les différentes horloges (CLK, CLK1 et FETCH) servant aux contrôles de séquencements indispensables pour l'exécution des instructions. L'unité de contrôle sélectionne la lecture, le décodage et l'exécution des instructions en réalisant diverses manipulations de contrôles dépendant de l'instruction, de l'état du processeur ainsi que des horloges.

Le jeu d'instructions du processeur contient 24 instructions regroupées en cinq catégories : les instructions arithmétiques et logiques, les instructions de branchements, les instructions d'accès à la mémoire, les instructions floues et les instructions d'interruptions logicielles. En moyenne, une instruction floue (MAX ou MIN) peut être émulée par 3.5 instructions dans un processeur généralisé. Donc les instructions floues de ce processeur RISC permettent d'optimiser les applications de cette logique en augmentant la vitesse jusqu'à un facteur de 2.5 par rapport aux processeurs ordinaires. Toutes les instructions ont un format fixe de 32 bits regroupés en cinq champs indiquant les deux opérandes de sources et un opérande de destination. Ces trois opérandes sont tous des registres, car le processeur utilise l'architecture à registres avec 3 opérandes. Ce format a permis d'obtenir un jeu d'instructions réduit à 24 instructions (voir fig. 1.1).

Les modes d'adressage utilisés par le processeur pour spécifier l'adresse d'un opérande sont les suivants : direct, indirect, immédiat, registre et inhérent. Dans ce format, le bit 16 spécifie le mode indirect, le bit 17, le mode registre et le bit 18, le mode direct. Pour sélectionner un mode d'adressage, un seul de ces trois bits est forcé à 1.

Le processeur peut fonctionner avec une fréquence d'horloge de 10 Mhz. Il a été conçu avec les outils de Mentor Graphics et le langage de description de matériel VHDL. Plusieurs outils d'aide à la conception ont été utilisés tels que QUICKSIM pour la simulation, Autologic pour la synthèse ainsi que la librairie des circuits FPGAs de la famille XILINX-4000. Ces circuits réutilisables, performants et économiques permettent d'obtenir un prototype très rapidement et ils sont reconfigurables.

Ce mémoire a aussi permis de présenter la conception d'un nouveau processeur flou qui comporte un mécanisme décisionnel en logique floue. Le processeur flou permet un

contrôle basé sur certaines règles de la logique floue et il fait une prise de décision adéquate en temps réel. Il utilise la règle de composition «max-min», la méthode du centre de gravité (COG) et la technique de consultation de tableau.

Dans un futur proche, l'ajout d'instructions floues capables de réaliser les autres opérateurs mentionnés au tableau 1.1 pourrait compléter le jeu d'instructions du processeur. Une structure d'exécution *pipelinée* à trois étages et des instructions vectorielles permettant de réaliser toutes les opérations d'un processeur flou sont également envisageables. La mise en oeuvre d'un prototype fonctionnel du nouveau processeur pourrait être effectuée avec des circuits FPGAs ( XC4025 ) de la famille XILINX-4000.

# BIBLIOGRAPHIE

Abrahams, M. S. et A Rushton. 1994. «Translation of VHDL for logic Synthesis». Microprocessors and Microsystems, vol. 18, octobre, p. 459-467.

Airiau, Rolland. 1990. VHDL : du langage à la modélisation. Lausanne : Presses polytechniques et universitaires romandes, 553 p.

Armstrong, J. R. 1989. Chip-Level Modeling With VHDL. Englewood Cliffs : Prentice-Hall.

Bouchon-Meunier, Bernadette. 1993. La logique floue. Paris : Presses universitaires de France, 127 p.

Brown, Franck. 1991. Processeurs RISC : l'exemple de l'AM29000. Paris : Masson, 164 p.

Brown, Stephen. 1992. Field-programmable gate arrays. Boston : Kluwer Academic, 206 p.

Camposano, Raoul et Wolf Wayne Hendrix. 1991. High-level VLSI synthesis. Boston : Kluwer Academic, 390 p.

Carini, PierPaolo. 1990. From a PLC to a PFC front end programming unit. Montréal : École polytechnique de Montréal, 36 f.

Chow, P. 1989. The MIPS-X RISC micrprocessor. Boston : Kluwer Academic Publishers.

Corder, R. J. 1989. «A High-Speed Fuzzy Processor». Proc. of 3nd IFSA Congress, août, p. 379-381.

Dancea, Ioan et Marchand Pierre. 1992. Architecture des ordinateurs. Boucherville, Québec : G. Morin, 513 p.

Dutton, Robert. 1991. «VLSI logic synthesis and design». Amsterdam : IOS Press, 318 p.

Etiemble, Daniel. 1991. Architecture des Processeurs RISC. Paris : A. Colin, 140 p.

Fox, S. et P. Gordon. 1991. The ASIC and Programming IC Strategy Report. Electronic Trend Publications, Saratoga, CA, USA.

Giovanni, Michelli et Ku David. 1992. High level synthesis of ASICs under timing and synchronisation constraints. Boston : Kluwer Academic, 294 p.

Gottwald, Sugfriend. 1993. Fuzzy set and Fuzzy logic : the foundations of applications, from a mathematical point of view. Toulouse : Teknea, 216 p.

Hamacher, V. C., Vranesic Z. G. et Zaky S. G. 1985. Structure des ordinateurs. Trad. de l'anglais par Daniel Etiemble et Michel Israël. Paris : McGraw-Hill, 455 p.

Hennessy, John et Patterson David A. 1994. Organisation et conception des ordinateurs : l'interface materiel/logiciel. Trad. de l'américain par Philippe Klein. Paris : Dunod, 660 p.

Hennessy, John L, Patterson David et Goldberg David. 1992. Architecture des ordinateurs : Une approche quantitative. Trad. de l'anglais par Daniel Etiemble et Michel Israël. Paris : McGraw-Hill, 751 p.

Heudin, Jean Claude et Panetto Christian. 1990. Architecture RISC : [Théorie et pratique des ordinateurs a jeux d'instruction réduit]. Paris : Dunod, 225 p.

Hoskins, Jim. 1992. IBM RISC system/6000 : a business perspective. New-York : J. Willey, 295 p.

IEEE Standard VHDL Language Reference Manual, IEEE Std. 1076-1987. The Institute of electrical and Electronics Engineers, Inc. 1988.

Jensen P. M. 1992. «Industrial application of Fuzzy logic control». Int Jounal Man-Machine Studies, vol. 12, p. 3-10.

Kane, Gerry et Heinrich Joe. MISP RISC architecture. Englewood Cliffs, N.J. : Prentice-Hall.

Kaufmann, Arnold. 1973.Introduction à la théorie des sous-ensembles flous à l'usage des ingénieurs. 4 t. Paris : Masson.

Katenis, Manopolis. 1985. Reduced instruction set computer architectures for VLSI. Ann Abor, Mich. : University Microfilms International, 214 p.

Katsumata, A., Jokumaga H. et S. Yasunobu. 1991. «Operation Method in Fuzzy Set Operation Processor». Proc. of ICCCD, octobre, p. 366-369.

Kickert, W. J. M. et E. H. Mandani. «Analysis of a Fuzzy logic Controller». Fuzzy Set Systems, vol. 1, p. 29-44.

Kickert, W. J. M. et Van Nauta Lemke. 1976. «Application of a Fuzzy Controller in a Warm Water Plant». Automatica, vol. 36, p. 301-308.

Krutz, Ronald. 1980. Microprossor and logic design. New-York : J. Willey, 467 p.

Lee, C. C. 1990. «Fuzzy Logic in Control Systems». IEEE Trans. on Systems, Man and Cybernetics. vol. 20 , No 2.

Leung, Steven. 1989. ASIC system design with VHDL : a paradigm. Boston : Kluwer Academic, 206 p.

Lippiatt, Arthur. 1980. Architecture des miniordinateurs et microprocesseurs. Trad. de l'anglais par Vincent Cogne. Paris : Eyrolles, 165 p.

Maeda Y. 1990. «Fuzzy obstacle Avoidance Method for a Mobile Robot Based on the Degree of Danger». Proc. of NAFIPS'90, juin, p. 169-172.

Mandani, E. H. 1974. «Aplication of fuzzy algorithms for the control of a dynamic plant». Proc. IEEE, vol. 12, p. 1385-1588.

Manzoul M. A. et D. Jayabharathi. 1992. «Fuzzy Controller on FPGA Chip». Proc. of IEEE International Conference on Fuzzy Systems, mars, p. 1309-1316.

Mazor, Standley. 1992. A guide to VHDL. Boston : Kluwer Academic, 307 p.

Mermet, Jean. 1992. VHDL for simulation synthesis and formal proofs hardware. Dorrecht, Pays-bas : Kluwer Academic, 307 p.

Mercouroff, Wladimir. 1990. Architecture matérielle et logicielle des ordinateurs et microprocesseurs. Paris : A. Colin, 258 p.

Miyamuto, M. et S. Yasunobu. 1984. «Predictive fuzzy control and its application to automatic train operation systems». First International Conference on Fuzzy Information Processing.

Naish, Paul. 1990. Conception des ASICs. Trad. de l'anglais par Francis Devos, Thierry Martin et Merigot Alain. Paris : Masson, 176 p.

Oudghiri, Houria et Kaminska Bozena. 1991. Synthèse de haut niveau : ordonnancement et allocation. Motréal : École polytechnique de Montréal, 65 p.

Perry, Douglas. 1994. VHDL. New-York : McGraw-Hill, 390 p.

Ristory, Joel et Ungaro Lucien. 1991. Cours d'architecture des ordinateurs : Conception des Circuits digitaux. t. 1. Paris : Eyrolles, 217 p.

Rose, J, A. E. Gamal et A. Sangiovanni-Vincentelli. 1993. «A Classification and Survey of Field-Programmable Gate Array Architectures». Proceeding of the IEEE, sept.

Sasao, Tsutomu. 1993. Logic synthesis and optimisation. Boston : Kluwer academic, 375 p.

Savaria, Yvon. 1988. Conception et vérification des circuits VLSI. Montréal : Éditions de l'École polytechnique de Montréal, 398 p.

Selz, M., Bartels S. et J. Syassen. 1993. «Proceedings of the VHDL Forum 1993», mars, p. 31-40.

Surmann, H., T. Tauber, A. Ungering et K. 1992. Goser. «Architecture of a Fuzzy Controller based on Field Programmable Gate Arrays». 2nd International Workshop on Field Programmable Logic and Applications, sept., p. 124-132.

Soumita, Dutta. 1993. «Fuzzy Logic Applications : Technological and Strategic Issues». IEEE trans. on engineering Management, vol. 40, No 3, août, p. 237-254.

Symon, J. R. et H. Watanabe. 1990. «Fuzzy Logic Inference Engine Board System». Proc. of International Conference on Fuzzy Logic and Neural Networks, juillet, p. 161-164.

Tabak, Daniel. 1991. Advanced microprocessors. New-York : McGraw-Hill, p 534.

Tabak, Daniel. 1987. Reduced Instruction Set Computer. Letchwood, Angleterre : Research Studies Press, 161 p.

Tocci, Ronald. 1992. Circuits numériques : Théorie et applications. Trad. de l'anglais

par André Lebel. Repentigny : R. Goulet, 776 p.

Trimberger, Stephen 1994. Field-prommable gate array technology. Boston : Kluver academic, 258 p.

Wallace C. S. 1964. «A suggestion for a fast mutiplier». IEEE trans. Electronic Computer, vol 13, février, p. 14-17.

Wanatabe H et D. Chen. 1993. «Evaluation of Fuzzy Instructions in a Risc Processor». IEEE International Conference on Fuzzy systems, p 521-526.

Watanabe, H, W. Detloff et K. Yount. 1990. «A VLSI Fuzzy Logic Controller with reconfigurable, Cascadable Architecture». IEEE Journal of Solid-State circuits, Vol. 25, No 2, avril, p. 376-382.

Watanabe, H. 1992. «A RISC Approch to Design of Fuzzy Processor Architecture». Proc. of IEEE International Conference on Fuzzy Systems, mars, p. 431-440.

Weyand, Dominique. 1991. Les processeurs RISC : Motorola 88000, R3000 de MIPS, Sparc de Sun. Paris : Eyrolles, 229 p.

Wong Y. K. et A. B. Rad. 1994. «Comparison of the performance of a fuzzy controller with a PID controller for a heating process». Microprocessors and Microsystems, vol. 18, septembre, p. 401-407.

XILINX Inc. : The Programmable Gate Array Data Book. 1991. Users Guide and Tutorial Book. San Jose/California.

Zadeh, L. A. et Kacprzyk Janusz. 1992. Fuzzy Logic for the management of uncertainty. New-York : J. Willey, 676 p.

Zadeh, L. A. 1973. «Outline of a New Approch to the Analysis of Complex Systems and Decision Processes». IEEE Transactions on systems, Man and Cybernetics, vol.SMC-3, vol. 1, janv., p. 28-32.

Zadeh, L. A. 1965. «Fuzzy set». Fuzzy set information and control, vol. 8, p. 338-353.

Zanella, Paola et Ligier Yves. 1993. Architecture et technologie des ordinateurs. Paris : Dunod, 425 p.