UNIVERSITE SAAD DAHLAB DE BLIDA

Faculté des sciences de l'ingénieur

Département d'Electronique

MEMOIRE DE MAGISTER

Spécialité : Micro-électronique

CONCEPTION ET IMPLEMENTATION D'UNE UNITE ARITHMETIQUE A PRECISION VARIABLE SUR CIRCUIT FPGA

Par

Messaoudi ABDELKRIM

Devant le jury composé de :

M. Djebari	**Maître de conférence, USD de Blida**	**Président**
K. Mazighi	**Maître de conférence, USTHB**	**Examinateur**
K. Achour	**Directeur de recherche, C.D.T.A**	**Examinateur**
H. Bougherira	**Chargée de cours, USD. de Blida**	**Examinatrice**
M. Anane	**Chargé de recherche, C.D.T.A**	**Rapporteur**
D. Naceur	**Chargée de cours, USD de Blida**	**Invitée**

Blida, octobre 2005

RESUME

Dans cette thèse, nous avons étudié la multiplication en précision variable, seul la mantisse a été traitée car elle représente la partie difficile à manipuler. Des performances en surface sont obtenues par l'exécution des opérations de multiplication et d'accumulation en parallèle, ce qui conduit à une réduction considérable des éléments nécessaire pour l'implémentation du multiplieur. La précision de calcul choisie dans ce travaille est comprise entre 1 et 64 mots. A cet effet, une implémentation hardware sur circuit FPGA de la famille Virtex-II a été réalisée. L'utilisation des ressources internes disponible sur ce type de circuit, tel que les blocs mémoires (SelectRam) pour le stockage des résultats intermédiaires et l'utilisation des blocs (DCM) pour la gestion des horloges, nous ont permet d'améliorer les performances de notre méthode. L'implémentation effectuée a montré que le temps de cycle exigé pour l'exécution de l'opération est de 33 ns.

Mots clefs : Norme IEEE-754, Multiplication, Précision variable, Circuit FPGA.

ABSTRACT

In this thesis, we have studied the multiplication in variable precision. Only the mantissa was treated because it represents the difficult part to manipulate. Surface performances are obtained by the execution of multiplications and accumulations in parallel, this leads to a considerable reduction of elements necessary for the multiplier implementation. The computation precision chosen in this work is included between 1 and 64 word. For that purpose, a harware implementation on Virtex-II FPGA circuits has been realized. The use of internal ressources of this type of circuit, such as the memory blocs (Select Ram) for the stocking of intermediate results and the use DCM blocs for the clock management, have improved the performances of our method. The implementation has shown that the time of cycle required for the execution of the operation is about 33ns.

Keywords: IEEE-754 format, Multiplication, Variable precision, FPGA circuits.

ملخص

في هذ ه الأطروحة، قمنا بدراسة عملية الضرب بدقة متغيرة، لقد تم معالجة الجزء العشري على انفراد لأنه يمثل الجزء الصعب في الدراسة. تمكنا من التقليص في المساحة بتنفيذ عمليا ت الضرب و التكدس على توازي مما أدى إلى تخفيض معتبر للعناصر الضرورية في إدماج الضارب. دقة الحساب في هذا العمل محصورة بين 1 و 64 كلمة (Mot).

لهذا الصدد تم إدماج على دارة منطقية قابلة للبرمجة (FPGA) من فصيلة Virtex-II. إن استعمال الموارد الداخلية المتوفرة في هذا النوع من الدارة، مثل جملة (DCM) من اجل تخزين النتائج الوسيطة و جملة(Select RAM) لإدارة التوقيت الزمني سمحت لنا بتحسين هذه الطريقة. مرحلة الإدماج المنجزة أثبتت أن زمن الدور اللازم لإجراء عملية الحسا ب يقدر بـ : 33 نانو ثانية.

الكلمات المفتاحية : قاعدةIEEE-754 ، الضرب، دقة متغيرة، دارةFPGA .

REMERCIEMENTS

Je remercie tout d'abord Dieu pour m'avoir aidé à réussir dans le concours de Magister, facilité mes études dans l'année théorique et enfin pour avoir guidé mes pas pour bien mener ce travail.

En second lieu, je tiens à exprimer mes remerciements les plus vifs a Monsieur H, BESSALAH, Directeur du centre de Développement des Technologies Avancées à Baba Hassen pour avoir mis tous les moyens nécessaires à l'aboutissement de cette thèse.

Mon remerciement et ma profonde gratitude à mon promoteur MR M. ANANE de m'avoir proposé ce sujet, de m' avoir encadré, pour ses remarques pertinentes et ses conseils judicieux qu' il m' a octroyé le long de mon travail.

Je remercie le président et les membres de jury, pour l'honneur qu'il m'on fait en consentant à juger mon travail.

Je voudrais aussi réserver une place de remerciements à l'équipe du laboratoire architecture des systèmes du centre CDTA, en particulier MR M. ISSAD.

Enfin, je tiens à remercier à toutes les personnes qui m'ont aidé de prés ou de loin durant l'élaboration de ce travail.

TABLE DES MATIERES

LISTE DES ILLUSTRATIONS, GRAPHIQUES ET TABLEAUX

INTRODUCTION

Depuis son apparition au lendemain de la deuxième guerre mondiale, l'informatique a connu un essor qui est peut être sans précédent dans l'histoire des sciences. Un micro-ordinateur de bureau est considérablement plus puissant que les monstres de 1950, qui tenaient dans les salles immenses et demandaient une maintenance des programmes effectuant des milliards d'opérations par second.

Très tôt, l'arithmétique des ordinateurs a évolué énormément. Ce développement est lié à l'apparition de l'arithmétique à virgule flottante ou plus simplement l'arithmétique flottante ; celle ci permet de représenter une plage de valeurs très grandes, aussi bien de très petits que de très grands nombres. Elle simplifie les calculs internes pour l'ensemble des opérations.

A la fin des années 70 chaque ordinateur avait sa propre représentation interne pour les nombres à virgule flottante. Or, l'arithmétique à virgule flottante possède certaines subtilités que le constructeur moyen ne maîtrisait pas forcément et certaines machines effectuaient certaines opérations de manière incorrecte. Pour remédier à cette situation, l'IEEE proposa un standard non seulement pour permettre les échanges de données en virgule flottante entre ordinateurs, mais aussi pour fournir aux constructeurs un modèle rodé, dont le fonctionnement était correct et maîtrisé. De nos jours, pratiquement tous les constructeurs ont des processeurs ou des coprocesseurs dédiés qui effectuent des calculs en virgule flottante et qui emploient la représentation au standard IEEE 754. Cette norme a constitué un grand progrès : elle a permis d'améliorer l'arithmétique réelle des ordinateurs et la portabilité des programmes.

Pour certaines applications telle que l'estimation des racines des polynômes et l'évaluation des fonctions élémentaires [3], les formats de simple précision (32 bits) ou double précision (64 bits) offerts par la norme IEEE 754 ne sont pas suffisants. Les approximations qu'elle impose lors des calculs entraînent des erreurs d'arrondis et des éliminations catastrophiques en s'accumulant au fil du programme conduisent à un résultat erroné. Afin de répondre aux exigences de précision et améliorer la qualité des résultats

d'un calcul, un format de longueur triple, quadruple ou en général à précision variable est nécessaire pour augmenter le nombre de chiffres significatifs est avoir une meilleure précision.

Les opérations arithmétiques en précision variable sont nécessairement effectuées en série car le traitement des nombres de taille quelconque d'une manière parallèle présente un problème au point de vue architecture puisque ce parallélisme nécessite des circuits de traitement de grande taille des opérandes à traiter, ce qui engendre une grande complexité de circuiterie. Comme le nombre de broche du boîtier des circuits est limité, les opérations à précision variable ne peuvent être exécutées qu'en série ou les opérandes sont introduits mot par mot.

La réalisation assez simple d'un circuit aussi complexe qu'un opérateur à précision variable ne peut que passer par une conception matérielle (description d'un circuit FPGA qui sont des circuits spéciaux programmables au niveau des portes logiques et permettant d'abaisser les coûts de productions), car l'emploi d'une conception logicielle exige des appels de subroutine consommatrices de temps, ce qui conduit à un temps d'exécution long [3]. Grâce à cette technologie nous allons implémenter dans ce travail une multiplication de deux nombres représentés en virgule flottante à précision variable sur circuit FPGA de Xilinx.

Notre travail est structuré en six chapitres :

- Dans le premier chapitre nous allons présenter les principes fondamentaux des nouvelles méthodes destinées à améliorer la qualité des résultats d'un calcul numérique.
- En deuxième chapitre, nous allons étudier l'arithmétique virgule flottante à précision variable en spécifiant son mode de représentation des nombres ainsi que le comportement de quelques opérations élémentaires.
- Dans le troisième chapitre, nous allons présenter l'algorithme de multiplication en virgule flottante à précision variable ainsi qu'un exemple de déroulement de cet algorithme.
- Le quatrième chapitre sera consacré à l'élaboration de l'architecture du multiplieur à précision variable en décrivant les différents étages constituant cet opérateur.

- Dans le cinquième chapitre, nous allons faire une description globale du circuit FPGA, et en particulier le circuit Virtex-II vu que celui-ci à été le support de l'implémentation de notre architecture.
- Dans le sixième chapitre, nous aborderons la dernière étape de la conception, qui est la description, la simulation et l'implémentation de notre architecture.

Et finalement nous terminons ce mémoire par une conclusion générale.

CHAPITRE 1

ARITHMETIQUE POUR UN CALCUL PRECIS ET EXACT

1.1. Introduction

L'arithmétique à virgule flottante est de loin la plus utilisée pour représenter les nombres réels sur ordinateur. Cette arithmétique présente indéniablement des qualités intéressantes, une grande dynamique (on peut représenter aussi bien de très grands que de très petits nombres). Le principal défaut de cette arithmétique est la propagation des erreurs d'arrondi qu'elle engendre à cause du nombre de chiffres limités de la mantisse.

L'accumulation des erreurs d'arrondi dans les calculs numériques et scientifiques utilisant des nombres en virgule flottante peut dégrader l'exactitude où invalider entierement les résultats. Comme le nombre d'opérations augmente, l'erreur numérique augmente ; dans quelques situations le résultat peut être totalement erroné à cause des petites erreurs numériques même après un très petit nombre d'opérations arithmétiques.

Dans le présent chapitre, nous allons donner tout d'abord un aperçu sur l'erreur d'arrondi et les erreurs dues à deux phénomènes : ***l'absorption*** et ***l'élimination*** introduite par l'arithmétique à virgule flottante. Ensuite nous présentons les diverses méthodes destinées à contrôler les erreurs numériques, ainsi que leurs applications potentielles.

1.2. Arithmétique à virgule flottante

1.2.1 Représentation

Les nombres à virgule flottante plus couramment appelés nombres flottants sont de la forme :

$$x = s \,.\, m \,.\, b^{e} \qquad (1.1)$$

Où s est le signe de x ($s = \pm 1$), b est la base (usuellement 2 ou 10), m la mantisse, et e l'exposant de x. La mantisse m comprend en général au plus p chiffres en base b ; on dit alors que p est la précision de x. L'exposant e est compris entre deux bornes : $e_{min} \leq e \leq e_{max}$.

1.2.2. La norme IEEE 754

La norme IEEE 754 (1985) définit 4 formats de flottants en base 2 : simple précision, simple précision étendue, double précision, et double précision étendue (extended en anglais). Les exposants ci-dessous sont relatifs à une mantisse $1 < m < 2$.

formats	tailles	précision	emin	emax	valeurs max
simple	32	23+1 bits	-126	+127	$3.403\ldots10^{38}$
double	64	52+1 bits	-1022	+1023	$1.798\ldots10^{308}$
extended	≥ 79	≥ 64	≤ -16382	≥ 16383	$1.190\ldots10^{4932}$

Tableau 1.1 : Formats offerts par la norme IEEE 754.

1.2.3. Erreurs de calcul arithmétique

1.2.3.1 Erreurs d'arrondi

Le système à virgule flottante a l'inconvénient d'être non clos arithmétiquement. En effet, si nous appelons S l'ensemble des nombres qui s'écrivent en virgule flottante dans un standard de notation (base, nombre de chiffres de mantisse et d'exposant, signe) fixé à l'avance, la somme, la différence, le produit et le quotient de deux éléments de S ne sont pas forcement un élément de S ; il faut donc les arrondir, c'est à dire les remplacer par des valeurs proches d'elles qui seront représentables exactement [1].

Chaque fois qu'une opération arithmétique est effectuée en machine, une erreur d'arrondi apparaît. Dans de nombreux calculs, ces erreurs d'arrondis n'ont pas de conséquences bien graves, mais il arrive parfois que leur cumul entraîne une erreur telle que le résultat n'ait plus aucun chiffre significatif, comme le montre l'exemple suivant [1] :

$$U_{n+1} = 111 - 1130/U_n + 3000/U_n U_{n-1}$$
$$U_0 = 2$$
$$U_1 = -4$$

il suffit de constater que :

$$U_n = (4\times5^{n+1} - 3\times6^{n+1})/(4\times5^n - 3\times6^n).$$

$$\lim_{n\to+\infty} Un = 6$$

En utilisant le logiciel MAPLE pour faire les calculs intermédiaires avec 100 chiffres décimaux on trouvera :

$$U_{90} = 99.999804031654181616165545845776784502796291953274$$

Alors que la valeur correcte de U_{90} est : 6.000000099684270640593864097 32

La limite de U_n est 6, pourtant, sur n'importe qu'elle machine on observe une convergence rapide de cette suite vers 100…

Des accumulations graves des erreurs d'arrondi se produisent également dans des applications réelles. Par exemple, un rapport du Général Accouting Office Américain, cité par Douglas Priest dans sa thèse de doctorat, analyse le comportement d'un missile anti-missile **«**patriot**»** durant la guerre du Golf. Ce missile n'avait pas réussi à intercepter sa cible à la suite d'accumulations d'erreurs d'arrondi, causant la mort de 28 personnes [2].

Plusieurs chercheurs ont essayé de trouver un moyen de contrôler ces erreurs d'arrondi. Nous allons par la suite présenter une méthode basée sur une nouvelle écriture des nombres ; elle consiste à représenter les réels par des intervalles.

1.2.3.2. Erreur d'élimination catastrophique

Ce phénomène se produit lorsqu'on soustrait des nombres très voisins. Les chiffres de poids forts s'éliminent et il faut alors normaliser le résultat en rajoutant des zéros en faible poids. Or, les opérandes sont en générale des résultats arrondis d'opérations antérieures et l'utilisation des opérandes exacts feraient apparaître des chiffres à la place de ces zéros. La valeur calculée peut alors être très différente de la valeur exacte.
Exemple d'élimination :

$X = 1.405\ 10^3$
$-Y = 1.404\ 10^3$
$Z = 0.001\ 10^3$

Ce qui donne après normalisation : $1.000\ 10^0$
Si X et Y résultent de calcul et sont les arrondis de :

X=1.405456 et Y =1.404012, le résultat exact est Z = 1.444.
L'erreur relative est ici de 40%.

1.2.3.3. Erreur d'absorption

Les erreurs d'absorption se produisent lors de l'addition ou la soustraction de deux nombres de grandeurs très différentes. Ceci est dû à la perte de chiffres lors de l'**alignement** des exposants. On peut même perdre tous les chiffres significatifs du plus petits des nombres de sorte que le résultat est simplement le plus grand des deux.
Exemple d'absorption :
$1.232 \times 10^4 + 2.104 \times 10^1$

$$\begin{array}{rl} 1.232 & 10^4 \\ + 0.002404 & 10^4 \\ \hline 1.234404 & 10^4 \end{array}$$

Ce qui après arrondi donne $1.234\ \ 10^4$

Autre exemple d'absorption (catastrophique) : soit à calculer $1.6 + 10^9 - 10^9$
Si on effectue le calcul selon l'ordre induit par le parenthèsage suivant : $1.6 + (10^9 - 10^9)$
on obtient : $1.6 + 0 = 1.6$, ce qui est normal.
Mais si on l'effectue selon l'ordre correspondant à : $(1.6 + 10^9) - 10^9$
on obtient : $10^9 - 10^9 = 0$, car par arrondi $1.6 + 10^9 = 10^9$
Cet exemple montre que l'addition avec les flottants est une opération ***non associative* !**

1.3. Solutions arithmétiques pour un calcul précis et exact

Les vingt dernières années ont vu l'éclosion de nombreuses techniques permettant d'améliorer considérablement la qualité des résultats d'un calcul. Parmi celles-ci, on distingue principalement les arithmétiques ;

- A précision variable.
- D'intervalles.
- D'intervalles à précision variable.

1.3.1. Arithmétique à précision variable

L'arithmétique à précision variable donne l'aptitude au programmeur de spécifier la précision de calcul basé sur le problème à résoudre et l'exactitude exigée des résultats. Dans la littérature, l'arithmétique à précision variable est aussi appelée arithmétique multi précision [3]. Elle est typiquement utilisée pour réduire les effets des erreurs d'arrondi et les éliminations catastrophiques dans les calculs numériques. En signalant que deux types de représentation des nombres sont utilisés :

- La représentation en virgule fixe à précision variable.
- La représentation en virgule flottante à précision variable.

1.3.1.1. La représentation en virgule fixe à précision variable

Dans un système en virgule fixe à précision variable, le nombre est d'habitude représentée par une série de chiffres, dans une base r. Le nombre de chiffres est référé à titre de longueur ou précision d'un nombre. Par exemple, un nombre en virgule fixe à précision variable X de longueur m est représenté par une série de chiffres x_0, x_1, … x_{m-1} tel que :

$$X=\sum_{i=0}^{m-1} x_i .r^{-i} \tag{1.2}$$

D'une manière typique, les chiffres sont soit des chiffres décimaux ou des entiers machine.

1.3.1.2. La représentation en virgule flottante à précision variable

Dans un système en virgule flottante à précision variable, le nombre est souvent représenté par un nombre en virgule fixe à précision variable, qui correspond à la mantisse, plus un exposant en virgule fixe. La longueur ou la précision d'un nombre en virgule flottante à précision variable est égale à la longueur de sa mantisse. Par exemple, un nombre en virgule flottante à précision variable de longueur n est représenté par une série de chiffres y_0, y_1, … y_{n-1} et un exposant e, tel que :

$$Y=b^e\sum_{i=0}^{n-1} y_i .r^{-i} \tag{1.3}$$

Ici, b est la base de l'exposent. Un nombre virgule flottante à précision variable Y est dit normalisé si le chiffre le plus significatif y_0 n'est pas nul.

Une seconde méthode pour la représentation des nombres en virgule flottante à précision variable est de représenté chaque nombre en virgule flottante à précision variable a titre d'une somme de série de nombres en virgule flottante à précision fixe. Dans ce cas, la longueur du nombre à précision variable est égale au nombre de nombres en virgule flottante à précision fixe. Avec cette représentation, un nombre en virgule flottante à précision variable Z est représenté à titre d'une série de p nombres en virgule flottante z_0, $z_1,\ldots z_{p-1}$ tel que :

$$Z=\sum_{i=0}^{p-1} z_i \qquad (1.4)$$

Chaque z_i est référé à titre d'un composant de Z pour fournir une précision garantie, il est souvent exigé que :

$$| z_0 | > | z_1 | > \ldots > | z_{p-1} |$$

Et que les exposants de chacune des deux valeurs consécutives à virgule flottante z_i et z_{i+1} diffèrent d'au moins l : où l est la longueur de la mantisse de z_i.

1.3.1.3. Applications

L'arithmétique en virgule fixe à précision variable est utile pour les applications telles que le cryptage et le décryptage. L'arithmétique à virgule flottante à précision variable est utile dans une variété d'applications scientifiques comprenant l'estimation de la racine des polynômes, l'évaluation des fonctions élémentaires etc.

1.3.2. Arithmétique d'intervalles

L'arithmétique d'intervalles spécifie une méthode précise pour l'exécution des opérations arithmétiques sur des intervalles. Elle produit deux valeurs pour chaque résultat. Les deux valeurs correspondent aux points extrêmes inférieur et supérieur d'un intervalle. La largeur de l'intervalle (la distance entre les deux points extrêmes) indique l'exactitude du résultat.

Si les points extrêmes d'un intervalle X sont dénotés par x_l et x_u respectivement. Par définition, un intervalle fermé $X = [x_l, x_u]$ se compose d'une série de nombres réels comprenant les deux points extrêmes x_l et x_u ($X= \{x : x_l \leq x \leq x_u \}$).

1.3.2.1. Opérations arithmétiques

Les quatre opérations arithmétiques de base (addition, soustraction, multiplication et division) sont définies pour deux intervalles $X=[x_l, x_u]$ et $Y= [y_l, y_u]$ comme suit :

$$Z = X + Y = [\, x_l + y_l, x_u + y_u \,] \quad (1.5)$$

$$Z = X - Y = [\, x_l - y_l, x_u - y_u \,] \quad (1.6)$$

$$Z = X * Y = [\min (x_l * x_l, x_l * y_u, x_u * y_l, x_u * y_u), \quad (1.7)$$
$$\max (x_l * y_l, x_l * y_u, x_u * y_l, x_u * y_u) \,]$$

$$Z = X / Y = [\min (x_l / y_l, x_l / y_u, x_u / y_l, x_u / y_u), \quad (1.8)$$
$$\max (x_l / y_l, x_l / y_u, x_u / y_l, x_u / y_u) \,]$$

D'autres opérations spéciales sont définies pour l'arithmétique d'intervalles ceux-ci comprennent l'intersection, l'union, la largeur et le point milieu qui sont définis comme suit :

$$\text{Itersect } (X, Y) = [\max. (x_l, x_l), \min (x_u, x_u)] \quad (1.9)$$

$$\text{Union } (X, Y) = [\min (x_l, y_l), \max (x_u, x_u)] \quad (1.10)$$

$$\text{Largeur } (X) = [x_u - x_l \,] \quad (1.11)$$

$$\text{Point milieu } (X) = (x_l + x_u) / 2 \quad (1.12)$$

Pendant l'exécution d'une opération arithmétique à intervalles en machine, un des points extrêmes d'un intervalle peut être non représentable. Dans ce cas, ce point est calculé par arrondissement extérieur. L'arrondissement extérieur exige que le point extrême inférieur soit arrondi vers une infinité négative et le point extrême supérieur soit arrondi vers une infinité positive. L'arrondissement extérieur assure que l'intervalle résultant inclut la vraie valeur.

Par exemple, si chaque intervalle est arrondi extérieurement en deux chiffres décimaux, alors :

$$[4.2, 4.4] + [1.4, 1.7] = [5.6, 6.1]$$

$$[4.2, 4.4] - [1.4, 1.7] = [2.8, 2.7]$$

$$[4.2, 4.4] \times [1.4, 1.7] = [5.88, 7.48] \approx [5.8, 7.5]$$

$$[4.2, 4.4] / [1.4, 1.7] = [2.47, 3.14] \approx [2.4, 3.2]$$

1.3.2.2. Fonctions élémentaires

L'arithmétique d'intervalles est aussi définie pour les fonctions élémentaires. Si une fonction élémentaire f(x) est monotone et croissante sur un intervalle X= $[x_l, x_u]$, l'intervalle résultant est f (x) = $[f(x_l), f(x_u)]$. Pour les fonctions monotones et décroissantes, l'intervalle résultant est $[f(x_u), f(x_l)]$. Pour les fonctions qui ne sont ni monotones ni croissantes ou décroissantes sur $[x_l, x_u]$, la fonction est évaluée à son point minimum et maximum, par exemple : Sin (0,2) = [0,1.0]
Puisque sin(x) a un point maximum de 1.0 a Π/2.

1.3.2.3. Applications

L'arithmétique d'intervalles est originalement proposée comme un outil pour limiter les erreurs d'arrondi dans les calculs numériques. Elle est aussi utilisée pour déterminer les effets des erreurs d'approximation et les erreurs qui sont dûs aux entrées non exactes. L'arithmétique d'intervalles est spécialement utilisée pour les calculs scientifiques, dans lesquelles, la donnée est inexacte ou peut prendre une large gamme de valeurs.

Depuis son apparition, l'arithmétique d'intervalles est appliquée pour résoudre les problèmes scientifiques tels que l'optimisation globale, l'évaluation des fonctions, l'estimation des racines des polynômes, etc.

1.3.3. Arithmétique d'intervalles à précision variable

L'arithmétique d'intervalles utilise des nombres en virgule flottante à précision fixe, afin de représenter chaque point extrême d'un intervalle. Cependant, ceci peut mener à des intervalles larges (wide interval), puisque chaque résultat est arrondi extérieurement. Avec l'arithmétique d'intervalles à précision variable, chaque point extrême est représenté par un nombre en virgule flottante à précision variable, ceci aide à empêcher les intervalles larges, dus aux erreurs d'arrondi et à l'élimination catastrophique, de se produire.

1.3.3.1. La représentation des nombres

Un intervalle à précision variable X est représenté par $[x_l, x_u]$, ou x_l et x_u sont des nombres à précision variable. Les points extrêmes de l'intervalle sont ensuite arrondis à la précision spécifiée.

Une autre méthode pour l'exécution de l'arithmétique d'intervalles à précision variable est de représenter chaque intervalle par un nombre à précision variable, plus un offset intervalle à précision fixe. Par exemple un intervalle à précision variable X est représenté par un nombre à précision variable x_{base}, plus un offset intervalle à précision fixe $[\underline{x}, \overline{x}]$. Les points extrêmes inférieurs et supérieurs sont calculés comme suit :

$$x_l = x_{base} + \underline{x} \tag{1.13}$$

$$x_u = x_{base} - \overline{x} \tag{1.14}$$

1.3.3.2. Applications

L'arithmétique d'intervalles à précision variable est utilisée pour résoudre plusieurs problèmes en informatique comprenant :

1. Optimisation globale.
2. Approximation des fonctions.
3. Détermination des racines d'un polynôme.
4. Exécution d'intégration et différentiation numérique.
5. Résolution des systèmes d'équations linéaires et non linéaires
6. Résolution du problème de la valeur initial.

L'arithmétique d'intervalles à précision variable à été appliquée à plusieurs domaines importants ; les systèmes de contrôle, de modélisation, de détection des défauts en VLSI, l'estimation des erreurs dans les systèmes laser, le contrôle des robots mobiles, etc.

1.4. Conclusion

Dans ce premier chapitre, nous avons exposé un bon nombre de techniques existantes permettant d'améliorer la qualité des résultats numériques ainsi que leurs applications. Dans le chapitre suivant, on s'intéresse d'une manière particulière à l'arithmétique virgule flottante à précision variable et on présente le format des nombres et le déroulement des trois opérations arithmétiques : addition, soustraction et multiplication.

CHAPITRE 2

ARITHMETIQUE EN VIRGULE FLOTTANTE A PRECISION VARIABLE

2.1. Introduction

Plusieurs calculs scientifiques et techniques sont numériquement intensifs et exigent des opérations en virgule flottante à précision élevée. Bien que la puissance de calcul des ordinateurs a beaucoup augmenté durant la dernière décennie, et ceci est dû au développement de la technologie VLSI, la précision des calculs des ordinateurs est restée presque inchangée, et souvent limitée à la simple et à la double précision de la norme IEEE-754. Dans quelques situations, l'arithmétique en virgule flottante ordinaire ne peut atteindre des résultats précis et exacts. Quelques applications exigent que certains calculs soient exécutés en haute précision, tels que le calcul du produit des matrices et la résolution des systèmes d'équations linéaires.

L'arithmétique en virgule flottante à précision variable est un outil utile pour surmonter les limitations numériques des ordinateurs courants. Elle permet à la précision de l'opérande d'être variable et basée sur l'exactitude exigée des résultats. Dans ce chapitre nous allons décrire ce modèle d'arithmétique.

2.2. Arithmétique en virgule flottante à précision variable

2.2.1. La représentation des nombres

Le format des nombres en virgule flottante à précision variable est représenté sur la figure 2.1. Il se compose d'un champ exposant (E), d'un bit signe (S), d'un champ type (T), d'un champ longueur de la mantisse (L), et d'une mantisse (M) qui comporte (L+1) mots à m bits (M (0) à M (L)). L'exposant est en format complément à 2. Le bit signe est à zéro si le nombre est positif et à un si le nombre est négatif. Le champ type indique si le nombre est fini, infini, nul, ou n'est pas un nombre. Le champ longueur spécifie le nombre de mots à m bits dans la mantisse [3].

Les mots de la mantisse sont stockés à partir du moins significatif M (0) au plus significatif M (L). La mantisse est normalisée (comprise entre 1/2 et 1).

La valeur d'un nombre en virgule flottante précision variable VP est :

$$VP = (-1)^S \times M \times 2^E \qquad (2.1)$$

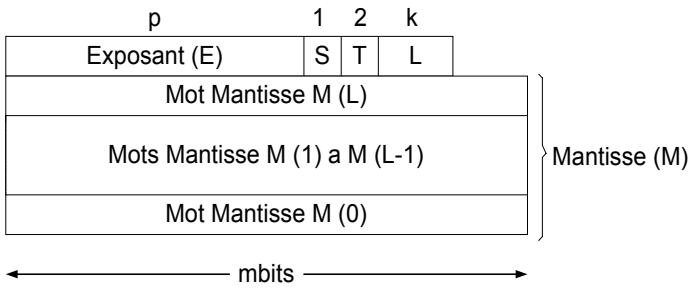

Figure 2.1 : Format en virgule flottante à précision variable.

2.2.2. Opérations arithmétiques [3]

2.2.2.1. Addition et soustraction

Pour l'addition en virgule flottante à précision variable R=A+B, les exposants des deux opérandes E_A et E_B sont comparés. S'ils sont égaux, on effectue l'addition des mantisses M_A et M_B en commencant par additionner les deux mots de rang faible M_A (0) et M_B (0) de façon à propager la retenue, ensuite additionner les deux mots de rang supérieurs M_A (1) et M_B (1) et la retenue précédente, on va ainsi du rang faible au rang fort jusqu'aux mots M_A (L) et M_B (L). Si une addition est exécutée sur des opérandes avec différents signes ou une soustraction est exécutée sur des opérandes avec le même signe, le nombre de plus petite valeur est soustrait du nombre de grande valeur et le signe du résultat est le même signe que celui du nombre de plus grande valeur.

La figure 2.2 montre une addition $M_R = M_A + M_B$ ou M_A et M_B sont deux nombres à 4 mots, le résultat M_R sera de 5 mots.

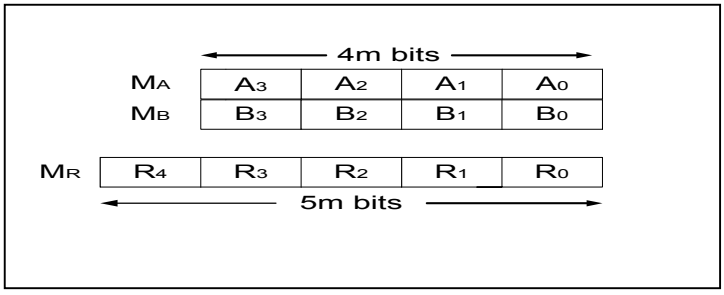

Figure 2.2 : Addition des mantisses en précision variable ($E_A = E_B$).

Si les exposants E_A et E_B ne sont pas égaux, on ne peut pas additionner ou soustraire directement les mantisses, ce problème est résolu par une opération de décalage à droite de la mantisse de l'opérande le plus petit en valeur absolue dans le but d'avoir un même exposant, c'est ce qu'on appelle *l'alignement des nombres*. Le nombre de décalage est égal à la différence entre les deux exposants E_D.

La figure 2.3 montre une addition $M_S = M_A + M_B$ ou M_A et M_B sont deux nombres à 4 mots, le résultat M_S sera de 6 mots.

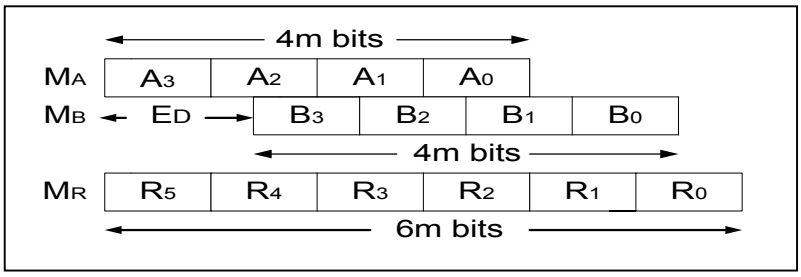

Figure 2.3 : Addition des mantisses en précision variable (E_A # E_B).

2.2.2.2. Multiplication

Pour la multiplication en virgule flottante à précision variable $R = A \times B$, les mantisses des deux opérandes M_A et M_B sont multipliées et les exposants E_A et E_B sont additionnés. Le signe du résultat est à zéro si les signes des deux opérandes S_A et S_B sont les mêmes, et à un s'ils sont différents.

La figure 2.4 indique la multiplication $M_R = M_A \times M_B$, ou M_A et M_B sont représentés sur trois mots. Le résultat M_R sera sur 6 mots. L'ordre de la génération et de l'accumulation des produits partiels est indiqué par les nombres entre parenthèses. Chaque paire sur cette figure correspondant à la multiplication d'un mot de M_B par M_A. Les mots de M_B sont abordés à partir du moins significatif au plus significatif (c.à.d, de B_0 à B_2). La retenue sortante d'une addition est utilisée comme retenue entrante dans la prochaine addition. Par exemple, après que la moitié la plus significatif du produit mot B_0A_0 est additionnée à la moitié la moins significatif du produit mot B_0A_1, la retenue sortante est utilisée comme retenue entrante quand la moitié la plus significatif du produit mot B_0A_1 est additionné à la moitié la moins significative de B_0A_2.

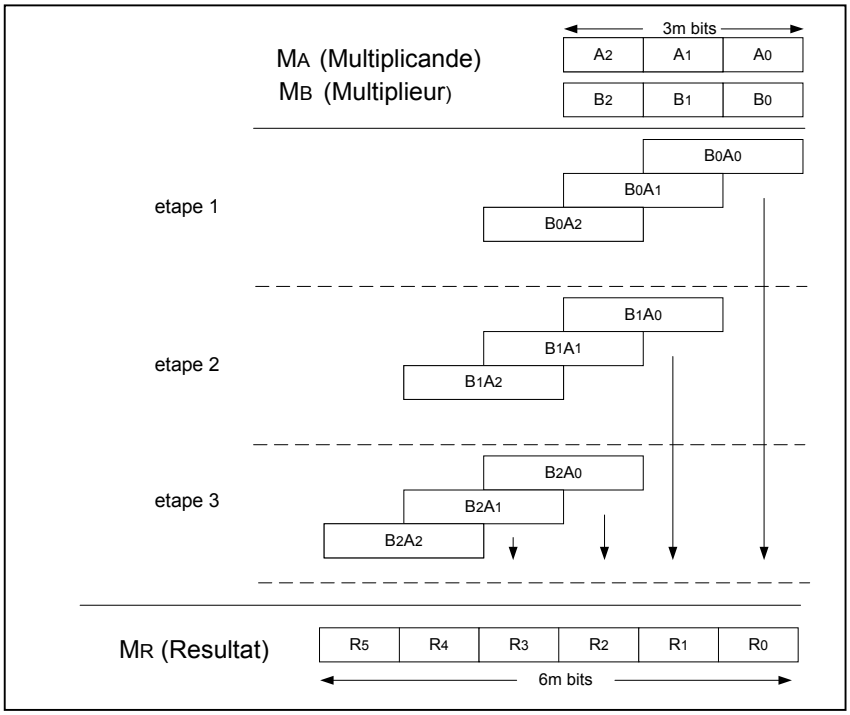

Figure 2.4 : Multiplication des mantisses en précision variable.

2.3. Solutions softwares et hardwares

Afin de surmonter les limitations numériques des systèmes ordinateurs existants, plusieurs solutions softwares et hardwares ont été développées pour l'arithmétique en virgule flottante à précision variable. Ces solutions augmentent l'exactitude des calculs en diminuant les effets de l'erreur d'arrondi et l'élimination catastrophique. Ils augmentent aussi la sûreté du calcul en produisant des résultats exacts.

2.3.1. Solution software

Depuis l'année 1960, un bon nombre de packages softwares ont été développés pour l'arithmétique virgule flottante à précision variable. Ces packages softwares utilisent des appels de subroutine pour exécuter les opérations arithmétiques virgule flottante à précision variable. Par exemple, l'opération à précision variable C=A+B est exécutée par la subroutine [4] :

MPADD(A, B, C)

Où A, B et C sont des nombres en virgule flottante à précision variable.

Les packages softwares à précision variable ont été développés pour une variété d'applications scientifiques comprenant l'estimation des racines des polynômes et l'évaluation des fonctions élémentaires.

Le package software, TRNSMP, translate les programmes FORTRAN dans des programmes à précision variable [5]. Un second package software, MPFUN, est ensuite utilisé pour exécuter les programmes à précision variable. Ces deux packages supportent des types de données à précision variable entier, réel et complexe[4].

Les packages softwares en virgule flottante à précision variable simulent les opérations arithmétiques avec des instructions intégrées dans la machine. Ils exigent des appels de subroutines consommatrices de temps pour chaque opération arithmétique, cela conduit à un temps d'exécution important. Par exemple une addition en hardware est a peu prés 1.000 fois plus rapide par rapport à une addition à précision variable utilisant un package software [6].

2.3.2.Soulution hardware

Afin de surmonter les limitations de vitesse des solutions softwares, plusieurs solutions hardwares, pour l'arithmétique virgule flottante à précision variable, ont été développées. Ceci inclut le processeur «CADAC» qui exécute les opérations arithmétiques virgule flottante à précision variable sur des nombres décimaux. Il effectue l'addition/soustraction, la multiplication, la division, la négation et la comparaison des nombres. Chaque nombre à précision variable se compose d'un bit signe (S), d'un drapeau d'étendu (X), d'un exposant (E), d'un descripteur de longueur (L) et d'une mantisse (M) qui comporte (2l+1) mots. Le descripteur de longueur indique la longueur de la mantisse (supérieur à 32 chiffres décimaux). Les longueurs les plus signifiantes de la mantisse sont indiquées par le drapeau d'étendu X.

La représentation des nombres du CADAC est indiquée par la figure 2.5. Chaque chiffre dans la partie mantisse est un nombre BCD [7].

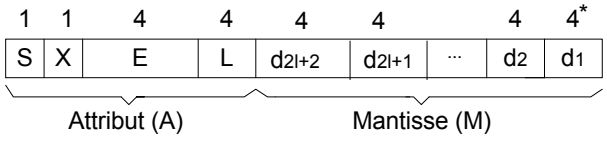

Figure 2.5 : La représentation d'un nombre flottant à précision variable du CADAC.

CADAC supporte cinq modes d'arrondi : arrondi au plus prêt, arrondi algébrique haut/bas et arrondi grandeur haut/bas. Il supporte aussi une variété de manipulation d'exception incluant le débordement, le dépassement, la division par zéro, l'indétermination, l'opération illégale, la virgule flottante invalide et non exacte.

2.4. Conclusion

Dans ce deuxième chapitre, nous avons présenté l'arithmétique en virgule flottante à précision variable qui consiste à manipuler des nombres d'une taille quelconque.

En deuxième partie, nous avons présenté les solutions softwares et hardwares liées à l'arithmétique en virgule flottante à précision variable.

Dans le chapitre suivant, nous allons décrire l'algorithme de la multiplication en virgule flottante à précision variable.

CHAPITRE 3

ALGORITHME DE MULTIPLICATION EN PRECISION VARIABLE

3.1. Introduction

Dans ce chapitre, nous présentons l'algorithme de la multiplication en précision variable, en vue d'une implémentation sur circuit FPGA. Les algorithmes de la multiplication (algorithme de Booth, réseau cellulaire,…etc.) ne peuvent traiter des opérandes de longueurs diverses : m, 2m, …nm bits et la taille des multiplieurs est proportionnelle au carré de la longueur des opérandes. Afin de réaliser un multiplieur à surface limitée indépendante de la taille de l'opérande, on a élaboré un algorithme de multiplication «multi-precision». Cet algorithme dicte la forme de l'architecture à concevoir pour notre application. Nous commençons par une description de la multiplication classique, puis nous expliquons les étapes du déroulement de notre méthode.

3.2. Multiplication classique en précision variable

La multiplication est un processus complexe par rapport à l'addition et la soustraction, son exécution nécessite un ensemble d'opérations (multiplication, stockage et addition). Prenons l'exemple de deux nombres A et B écrits chacun sur 3 mots de m bits tels que :

$A = A_2\ A_1\ A_0$ (Multiplicande)

$B = B_2\ B_1\ B_0$ (Multiplieur)

et $R = A \times B$ (Produit)

Si nous effectuons la multiplication de $A = A_2\ A_1\ A_0$ et $B = B_2\ B_1 B_0$ par la méthode classique (voir figure 3.1). On aura à multiplier tout d'abord le mot de plus faible poids du multiplieur (B_0) par chaque mot du multiplicande A, cela donne le premier produit partiel PP1, on stocke ce produit partiel dans une mémoire M, le second produit partiel PP2 est obtenu par la multiplication de B_1 par A que l'on stocke en mémoire M. On calcule ensuite de la même façon le troisième produit partiel PP3 que l'on stocke une nouvelle fois dans la

mémoire M. L'addition de tous ces produits partiels donne le résultat de la multiplication $R= R_5R_4R_3R_2R_1R_0$.

La multiplication A×B (A et B écrits chacun sur n×m bits) par la méthode classique, peut être résumée en trois tâches principales :

1- La multiplication de chaque mot du multiplicande A par chaque mot du multiplieur B en générant (n) produits partiels.
2- L'enregistrement des produits partiels générés dans une mémoire.
3- L'addition totale des produits partiels mémorisés.

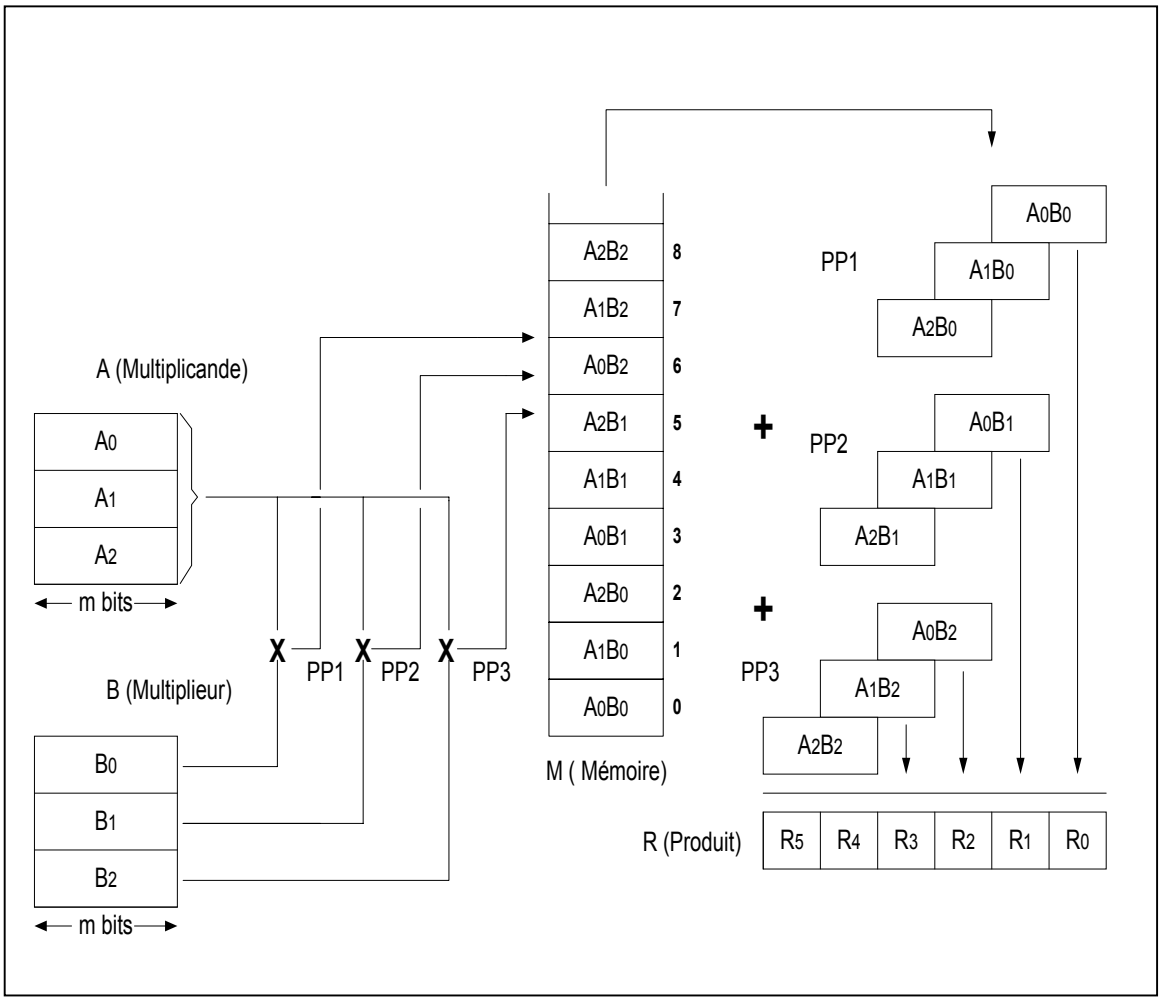

Figure 3.1 **:** Déroulement de la multiplication classique en précision variable pour deux nombres de 3 mots à m bits.

3.3. Présentation de l'algorithme

La méthode classique de multiplication en précision variable présente un inconvénient du fait que pour des opérandes écrits sur un grand nombre de mots (n) on aura à utiliser une mémoire de grande taille (d'une capacité n×n mots à 2m bits), ce qui engendre malheureusement une complexité en surface du circuit à réaliser. Pour remédier à cet inconvénient, on a utilisé un processus qui consiste à effectuer l'accumulation des produits partiels au fur et à mesure qu'ils sont générés (voir figure 3.2), cela en faisant l'addition du produit "mot précédent" et du produit "mot suivant" en portant le résultat en mémoire, ensuite additionner le prochain produit mot avec le résultat mémorisé ; ce même processus se déroulera pour chaque couple de mot du multiplieur et du multiplicande jusqu'au dernier et on aura à utiliser simplement une mémoire de taille (2n) mots à m bits (qui servira à stocker les résultats intermédiaires W_K et les résultats finals R_P de la multiplication) au lieu de n^2 mots à 2m bits.

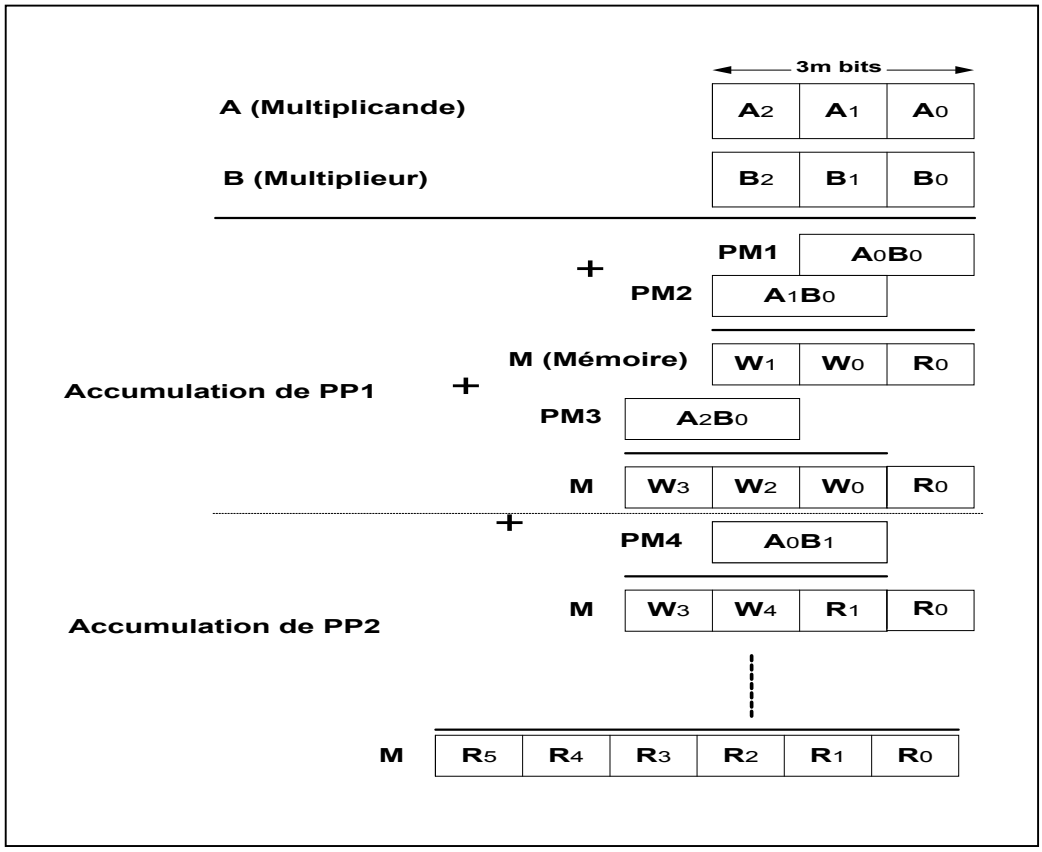

Figure 3.2 **:** Processus d'accumulation des produits partiels lors de la multiplication de deux nombres de 3 mots à m bits.

En fait les additions des produits mots A_iB_j ne se font pas arbitrairement, mais en respectant le poids affecté à la partie inférieure (les m bits les moins significatifs de A_iB_j) et à la partie supérieure (les m bits les plus significatifs de A_iB_j) de chacun suivant sa position dans la suite des additions (voir figure 3.3).

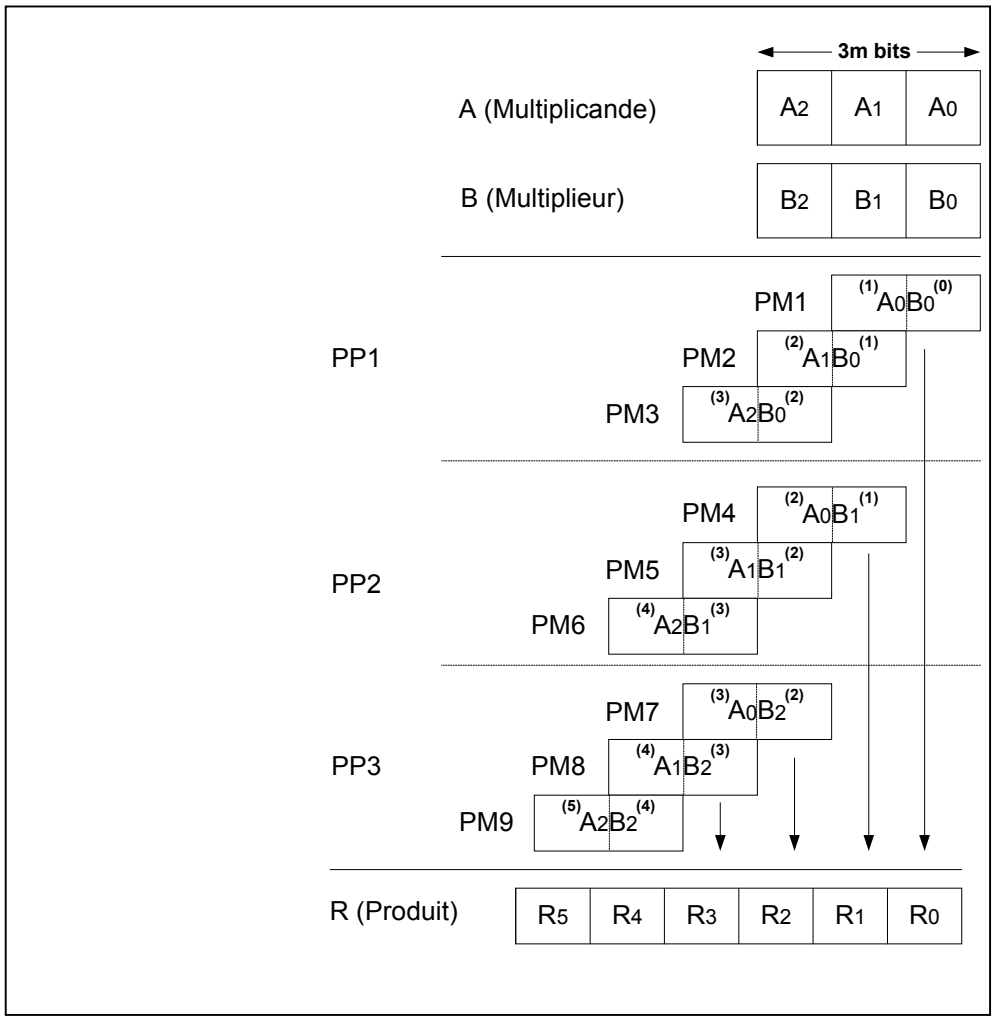

Figure 3.3 **:** Disposition des produits partiels à additionner selon le poids affecté à la partie inférieure et supérieure d'un produit mot..

Cela veut dire que lors de chaque itération de la multiplication (voir figure 3.4), il faudra additionner la partie inférieure du produit mot A_iB_j portant le poids (i+j) au résultat intermédiaire logé en mémoire M a l'adresse (i+j). Le résultat obtenu sera rangé à la même adresse (i+j) et sera pris soit comme résultat final ou intermédiaire qui doit être additionné à la partie inférieure ou supérieure du prochain produit mot portant le même poids (i+j).

Même opération pour la partie supérieure portant le poids (i+j+1) ; elle sera additionnée au résultat intermédiaire rangé en mémoire M à l'adresse (i+j+1), le résultat obtenu sera rangé à la même adresse (i+j+1) et sera pris soit comme résultat final ou intermédiaire, qui doit être additionné à la partie inférieure ou supérieure du prochain produit mot portant le même poids (i+j+1).

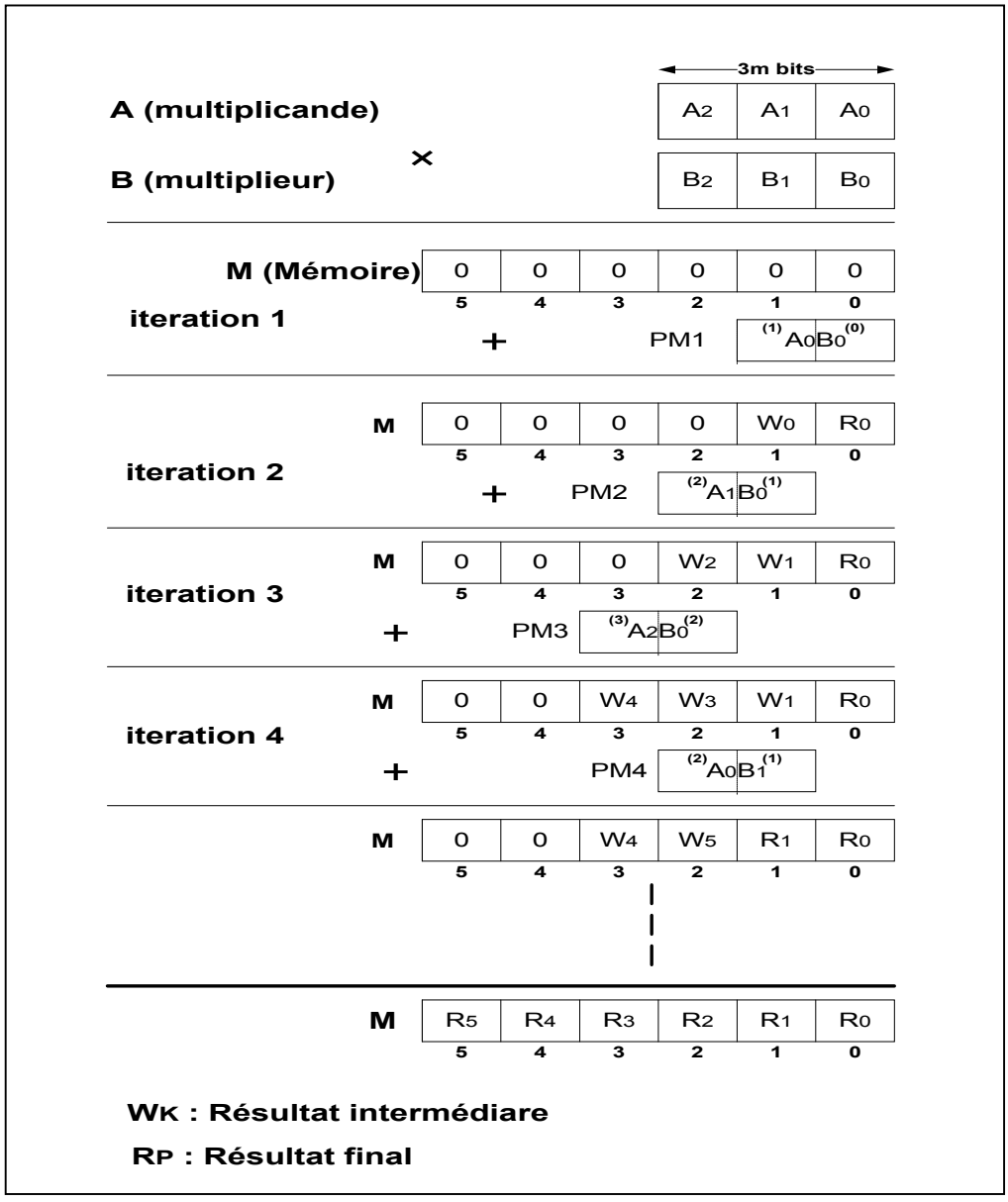

Figure 3.4 : Différentes phases de l'algorithme de multiplication de A par B (deux nombres de 3 mots à m bits).

3.4. Conclusion

Dans ce troisième chapitre, nous avons présenté l'algorithme de multiplication en précision variable. Cet algorithme multiplie chaque mot du multiplicande A par chaque mot du multiplieur B séparément, le produit résultant R est composé de (2n) mots de m bits et exige $(n)^2$ multiplications, chacune d'elles est sur (m×m) bits ainsi que $2(n)^2$ additions, chacune d'elles est sur m bits.

L'algorithme élaboré présente deux avantages : le premier avantage est que les additions des produits partiels se font coup par coup au lieu d'être faites globalement à la fin, ce qui se traduit par une simple utilisation d'un seul module d'addition qui va servir à nouveau pour chaque couple de mots du multiplicande et du multiplieur jusqu'au dernier. Le deuxième avantage est que nous aurons à utiliser simplement une mémoire d'une petite taille ((2n) mots à m bits au lieu de (n^2) mots à 2m bits) qui va servir à stocker les résultats intermédiaires et finaux de la multiplication, ce qui présente une réduction considérable des ressources utilisées pour l'implémentation de notre architecture. Cette réduction est d'autant plus importante que la taille des mots traités est grande.

La conception de l'architecture permettant d'exécuté cet algorithme sera présenté dans le chapitre suivant.

CHAPITRE 4

ARCHITECTURE DU MULTIPLIEUR A PRECISION VARIABLE

4.1. Introduction

Après avoir présenté l'algorithme de multiplication des mantisses de deux nombres écrits en virgule flottante à précision variable, l'étape suivante consiste à concevoir l'architecture qui réalise cet algorithme. Tout d'abord nous présentons à travers un synoptique l'architecture globale du multiplieur à précision variable. Puis nous aborderons la description de la micro-architecture en donnant un exemple de déroulement de l'algorithme dans celle-ci.

4.2. Architecture globale du multiplieur à précision variable

L'architecture que nous allons présenter est subdivisée en cinq modules :

1. Un module d'adressage
2. Un module de multiplication
3. Un module d'accumulation
4. Un module générateur des signaux d'horloge
5. Un module générateur des signaux de commande

Le synoptique de celle-ci est illustré sur la figure 4.1.

Le rôle attribué au module d'adressage est le calcul des adresses mémoires pendant la phase de traitement et la phase d'expédition. Ce module génère en sortie :

- L'adresse X de la mémoire multiplicande MA et l'adresse Y de la mémoire multiplieur MB pour la lecture des données pendant la phase de traitement.
- L'adresse Z de la mémoire résultat MR d'une part pour la lecture/écriture des résultats intermédiaires pendant la phase de traitement et d'autre part pour la lecture des résultats finaux pendant la phase d'expédition.

Les signaux à l'entrée du module d'adressage ; RST, H, H_1, H_2, L_C, K_1, K_2 et K_4 sont définis dans la section 4.3.1.

.

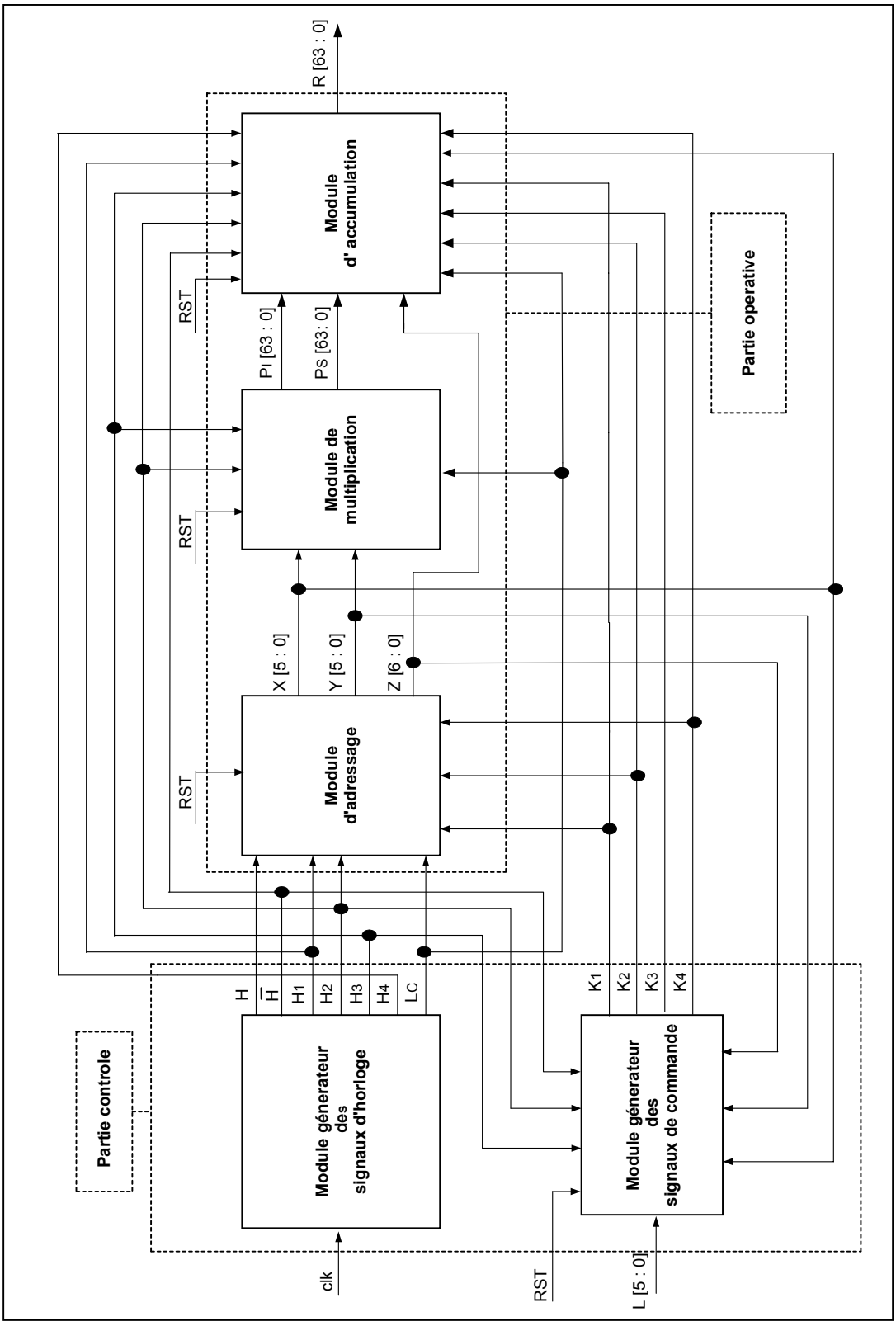

Figure 4.1 : Synoptique de l'architecture globale du multiplieur à précision variable

Le rôle attribué au module de multiplication est d'effectuer pendant chaque séquence du cycle de calcul la multiplication d'un mot A_i qui se trouve dans la mémoire MA à l'adresse X par un mot B_j qui se trouve dans la mémoire MB à l'adresse Y, et partagé le produit mot résultant P en deux parties égales ; la partie inférieure P_I et la partie supérieure P_S.
Les signaux à l'entrée du module de multiplication ; L_C, H_2 et H_3 sont définis dans la section 4.3.2.
Le rôle attribué au module d'accumulation est d'effectuer pendant chaque séquence du cycle de calcul :

- L'addition de la partie inférieure du produit P_I fourni par le module de multiplication et le résultat intermédiaire W_1 qui se trouve dans la mémoire MR à l'adresse Z=X+Y, ensuite le stockage du résultat S_1 à la même adresse (X+Y).
- L'addition de la partie supérieure du produit P_S fourni par le module de multiplication et le résultat intermédiaire W_2 qui se trouve dans la mémoire MR à l'adresse Z=X+Y+1, ensuite le stockage du résultat S_2 à la même adresse (X+Y+1).

Les signaux à l'entrée du module d'accumulation ; $\overline{H}$, H_1, H_2, H_3, H_4, L_C, K_1, K_2, K_3, K_4 sont définis dans la section 4.3.3.
Le module générateur des signaux d'horloge génère à partir de l'horloge externe clk six signaux d'horloge ; H, $\overline{H}$, H_1, H_2, H_3 et H_4 permettant de contrôler les transferts de données a travers les éléments de la logique combinatoire constituant les modules (d'adressage, de multiplication et d'accumulation), ainsi qu'un signal de control L_C permettant d'activer ces signaux après 346 cycles de clk .
Le module générateur des signaux de commande effectue la comparaison des deux adresses X, Y et l'adresse extrême L des deux mémoires MA et MB, ainsi que la comparaison de l'adresse Z et l'adresse extrême 2L+1 de la mémoire MR, en fournissant en sortie les signaux de commande K_1, K_2, K_3 et K_4 aux éléments de la logique combinatoire et de la logique séquentielle constituant les modules (d'adressage, de multiplication et d'accumulation) pour qu'elles puissent réaliser ces propres fonctions.

Dans ce qui suit, nous aborderons la description de la micro-architecture de chaque module constituant le multiplieur à précision variable.

4.3. Micro architecture du multiplieur à précision variable

4.3.1. Module d'adressage

Comme il est indiqué sur la figure 4.2, ce module est constitué de cinq circuits :

1. Un circuit d'adressage de la mémoire multiplicande
2. Un circuit d'adressage de la mémoire multiplieur
3. Un circuit d'adressage de la mémoire résultat en phase de traitement
4. Un circuit d'adressage de la mémoire résultat en phase d'expédition
5. Un circuit de sélection d'adresses

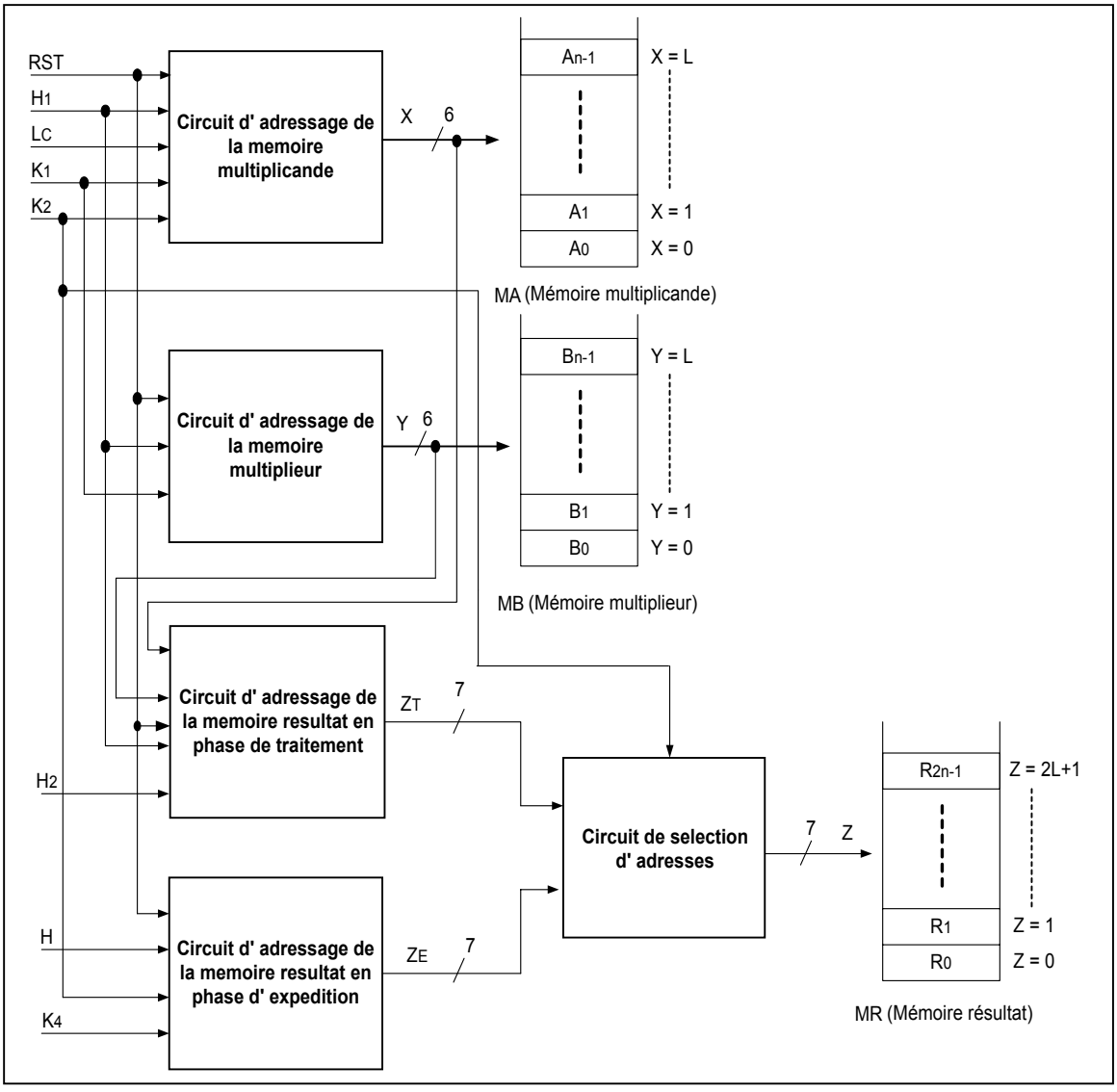

Figure 4.2 : Synoptique de la micro architecture du module d'adressage.

Le circuit d'adressage de la mémoire multiplicande génère en sortie l'adresse X permettant l'adressage de la mémoire MA pendant la phase de traitement. Il reçoit en entrée :

- La commande RST permettant de positionner X à 0 au début du cycle de calcul, et cela pour accéder au mot de poids le plus faible (A_0) qui est rangé en mémoire MA.
- L'horloge H_1 permettant de décaler X d'une unité (X+1) lors de chaque séquence du cycle de calcul, et cela pour accéder au mot de poids supérieur (A_{i+1}) qui se trouve dans la mémoire MA.
- Le signal L_C permettant d'activer l'horloge H_1 après 346 cycles de clk.
- La commande K_1 permettant de positionner X à 0 lors de la multiplication du jième mot du multiplieur (B_j) par le mot de poids le plus fort du multiplicande (A_{n-1}), et cela pour accéder à nouveau au mot de poids le plus faible (A_0) qui se trouve dans la mémoire MA.
- La commande K_2 permettant de désactiver l'horloge H_1 lors de la multiplication du mot de poids le plus fort du multiplieur (B_{n-1}) par le mot de poids le plus fort du multiplicande (A_{n-1}), et cela pour arrêter le processus de multiplication A×B.

Le circuit d'adressage de la mémoire multiplieur génère l'adresse Y permettant l'adressage de la mémoire MB pendant la phase de traitement. Il reçoit en entrée :

- La commande RST permettant de positionner l'adresse Y à 0 au début du cycle de calcul, et cela pour accéder au mot de poids le plus faible (B_0) qui se trouve dans la mémoire MB.
- L'horloge H_1 permettant de décaler l'adresse Y d'une unité (Y+1), et cela pour accéder au mot de poids supérieur (B_{j+1}) qui se trouve dans la mémoire MB.
- La commande K_1 permettant d'activer l'horloge H_1 lors de la multiplication du jième mot du multiplieur (B_j) par le mot de poids le plus fort du multiplicande (A_{n-1}).

Le circuit d'adressage de la mémoire résultat en phase de traitement génère à partir des deux adresses X et Y fournies par les circuits précédents l'adresse Z_T permettant l'adressage de la mémoire résultat MR pendant la phase de traitement.

Le circuit d'adressage de la mémoire résultat en phase d'expédition génère l'adresse Z_E permettant l'adressage de la mémoire résultat MR pendant la phase d'expédition. Il reçoit en entrée :

- L'horloge H permettant à chaque séquence du cycle d'expédition de décalé Z_E d'une unité (Z_E+1) pour accéder au mot de poids supérieur (R_{p+1}) qui se trouve dans la mémoire MR.
- La commande K_2 permettant d'activer l'horloge H à la fin du cycle de calcul.
- La commande K_4 permettant de désactiver l'horloge H à la fin du cycle d'expédition.

Le cinquième circuit du module d'adressage permet l'adressage de la mémoire résultat. Il reçoit en entrée la commande K_2 permettant de sélectionner ou bien l'adresse Z_T pendant la phase de traitement ou Z_E pendant la phase d'expédition.

Dans ce qui suit, nous allons présenter l'architecture de chaque circuit constituant le module d'adressage.

4.3.1.1. Circuit d'adressage de la mémoire multiplicande

Comme il est illustré sur la figure 4.3, ce circuit est constitué de deux parties :

- La première partie est composée d'un compteur CMPTA considéré comme générateur de l'adresse X, celui-ci s'incrémente au front montant de l'horloge H_1.
- La seconde partie est composée des deux circuits logiques L_1 et L_2 qui génèrent respectivement les signaux S_1 et S_2 connectés respectivement aux entrées CL (clear) et CE (clock enable) de CMPTA, permettant la remise à zéro de ce compteur et l'activation de l'horloge H_1.

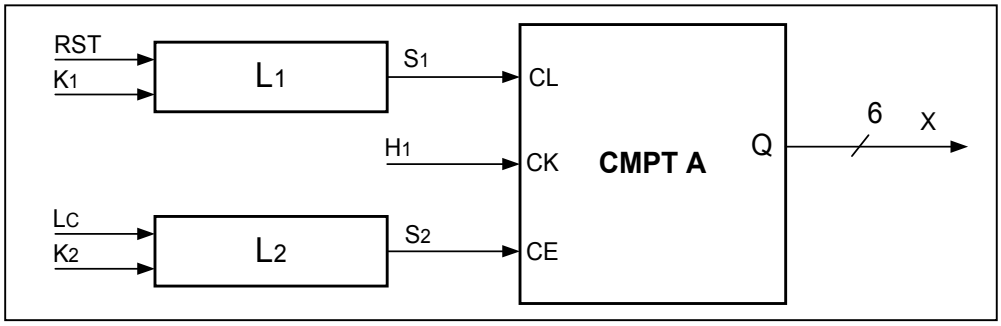

Figure 4.3 : Synoptique du circuit d'adressage de la mémoire multiplicande.

Les tables de vérité présentées sur les tableaux 4.1(a) et 4.1(b) résument le fonctionnement des deux circuits L_1 et L_2.

K_1	RST	CL	S_1
0	0	0	0
0	1	1	1
1	0	1	1
1	1	X	X

-a-

K_2	L_C	CE	S_2
0	0	0	0
0	1	1	1
1	0	0	0
1	1	X	X

-b-

Tableaux 4.1 (a-b) : Etats du signal de sortie des circuits L_1 et L_2.

Ceux-ci nous permettent de déduire les expressions suivantes :

$S_1 = RST + K_1$

$S_2 = L_C \cdot \overline{K_2}$

Nous avons traduit ces deux expressions en deux circuiteries représentés sur la figure 4.4.

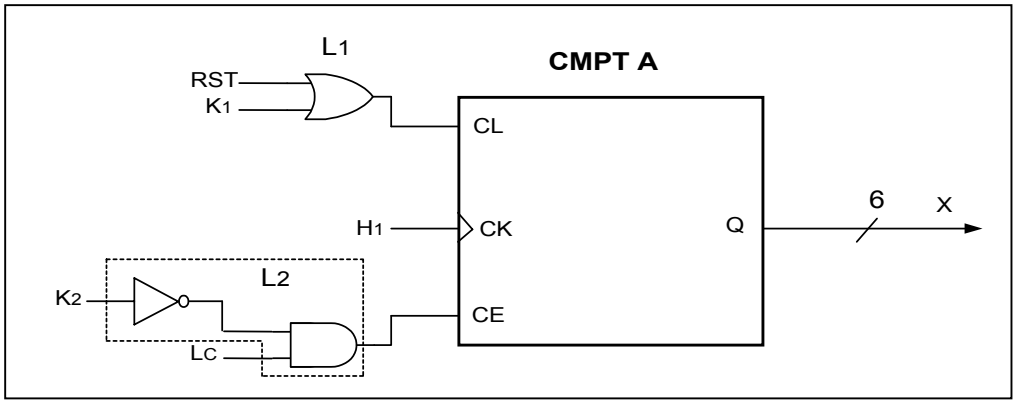

Figure 4.4 : Circuit d'adressage de la mémoire multiplicande.

4.3.1.2. Circuit d'adressage de la mémoire multiplieur

Comme il est indiqué sur la figure 4.5, ce circuit est constitué d'un compteur CMPTB considéré comme générateur de l'adresse Y, celui-ci s' incrémente au front montant de l'horloge H_1. Les signaux RST et K_1 connectés respectivement aux entrées CL (clear) et CE (clock enable) de CMPT B, permettent la remise à zéro de ce compteur et l'activation de l'horloge H_1.

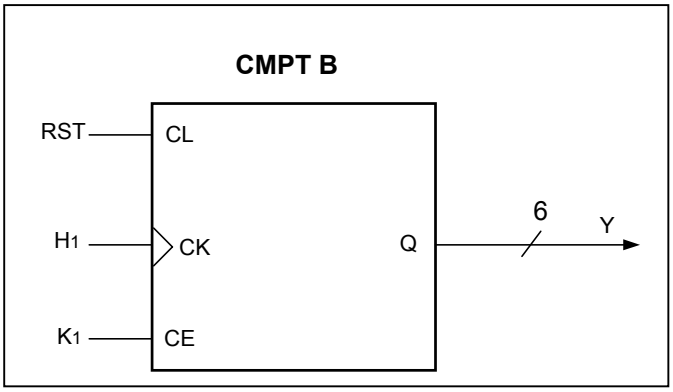

Figure 4.5 : Circuit d'adressage de la mémoire multiplieur.

4.3.1.3. Circuit d'adressage de la mémoire résultat en phase de traitement

Comme il est illustré sur la figure 4.6 ce circuit est constitué de quatre parties :

- La première partie est composée de l'additionneur ADD1 considéré comme générateur de l'adresse (X+Y), celui-ci effectue l'addition des deux adresses X et Y.
- La deuxième partie est composée de l'additionneur ADD2 considéré comme générateur de l'adresse (X+Y+1), celui-ci effectue l'addition des deux adresses X, Y et en y' rajoutant 1.
- La troisième partie est composée des deux registres L_1 et L_2 permettant de mémoriser respectivement les adresses (X+Y) et (X+Y+1) au front montant de l'horloge H_2.
- La quatrième partie est composée du multiplexeur MUX commandé par la sortie S du circuit logique L_A. Il permet la sélection de l'adresse (X+Y) quand S=0 et la sélection l'adresse (X+Y+1) quand S=1.

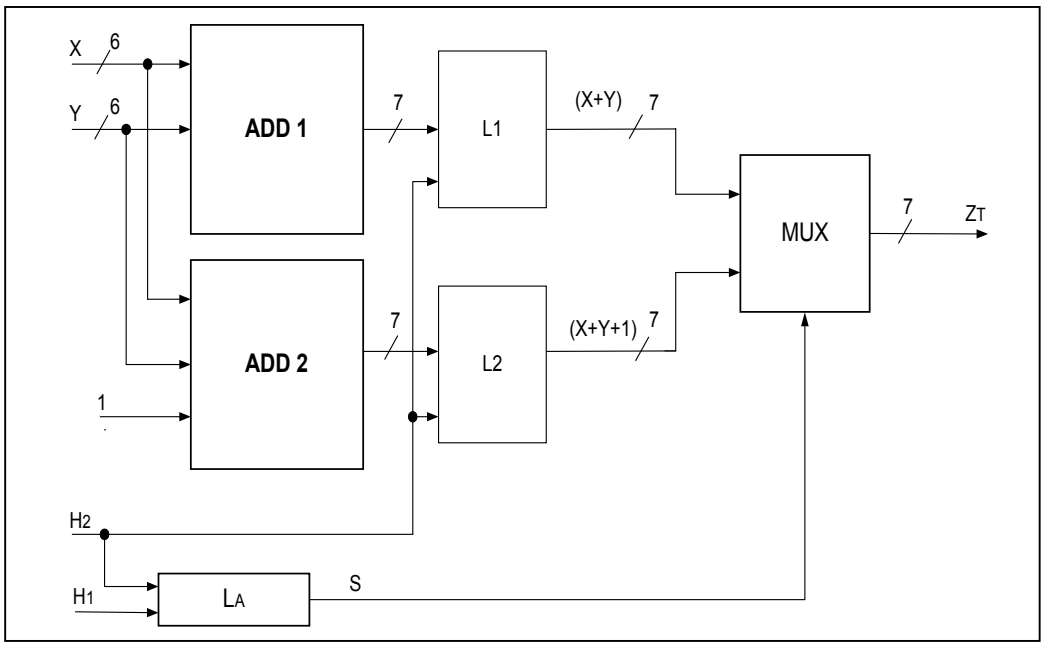

Figure 4.6 : Synoptique du circuit d'adressage de la mémoire résultat en phase de traitement.

La table de vérité présentée sur le tableau 4.2 résume le fonctionnement du circuit L_A.

H_2	H_1	S	Z_T
0	0	0	X+Y
0	1	1	X+Y+1
1	0	1	X+Y+1
1	1	0	X+Y

Tableaux 4.2 : Etats du signal de sortie du circuit L_A.

Ceux-ci nous permettent de déduire l'expression suivante :

$S = H_1 \times \overline{H_2} + \overline{H_1} \times H_2$

Nous avons traduit cette expression à une circuiterie représentée sur la figure 4.7.

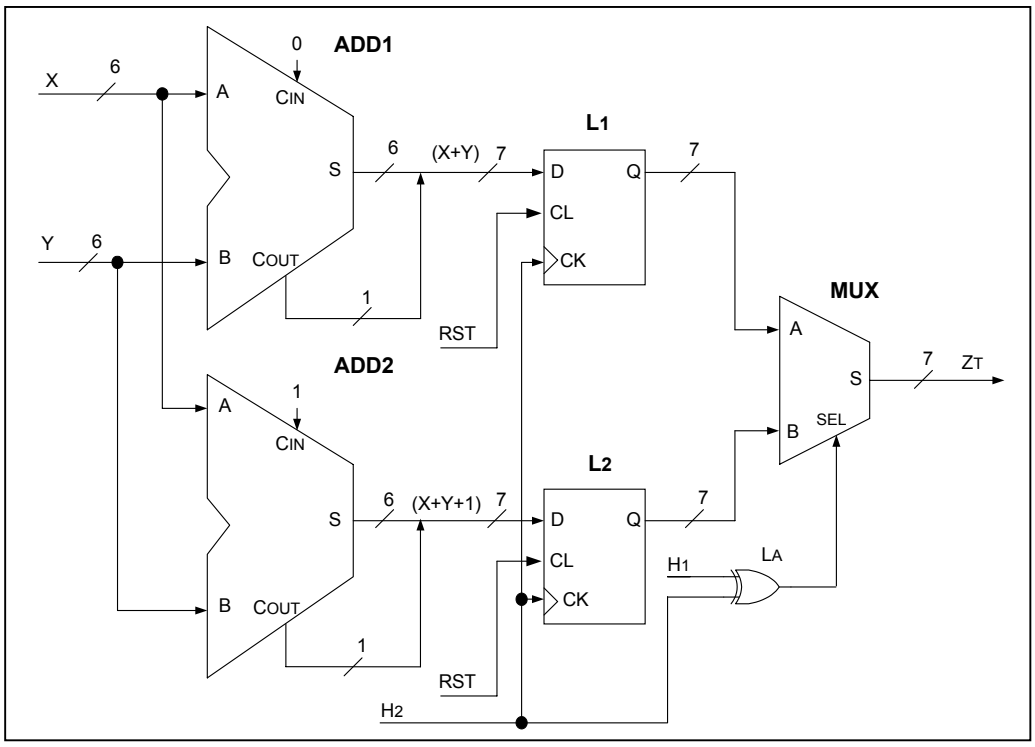

Figure 4.7 : Circuit d'adressage de la mémoire résultat en phase de traitement.

4.3.1.4. Circuit d'adressage de la mémoire résultat en phase d'expédition

Comme il est indiqué sur la figure 4.8, ce circuit est constitué de deux parties :

- La première partie est composée du compteur CMPTC considéré comme générateur de l'adresse Z_E, celui-ci est remet a zéro par la commande RST et s' incrémente au front montant de l'horloge H.
- La seconde partie est composée du circuit logique LB qui génère en sortie le signal S connecté à l'entrée CE (clock enable) de CMPTC permettant l'activation de l'horloge H.

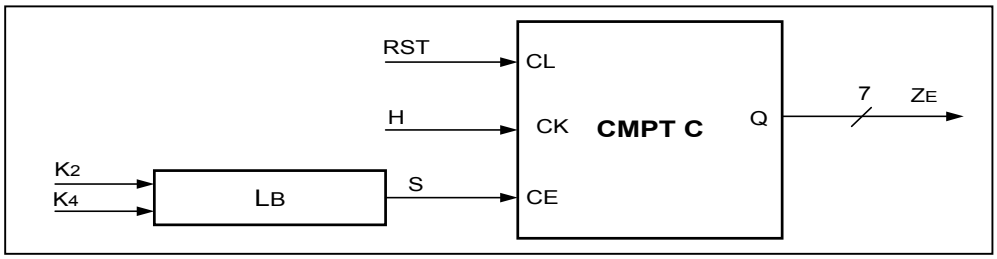

Figure 4.8 : Synoptique du circuit d'adressage de la mémoire résultat en phase d'expédition.

La table de vérité présentée sur le tableau 4.3 résume le fonctionnement du circuit LB.

K_4	K_2	CE	S
0	0	0	0
0	1	1	1
1	0	0	0
1	1	X	X

Tableau 4.3 : Etats du signal de sortie du circuit LB.

Ceux-ci nous permettent de déduire l'expression suivante : $S = K_2 \times \overline{K_4}$

Nous avons traduit cette expression à une circuiterie représenté sur la figure 4.9.

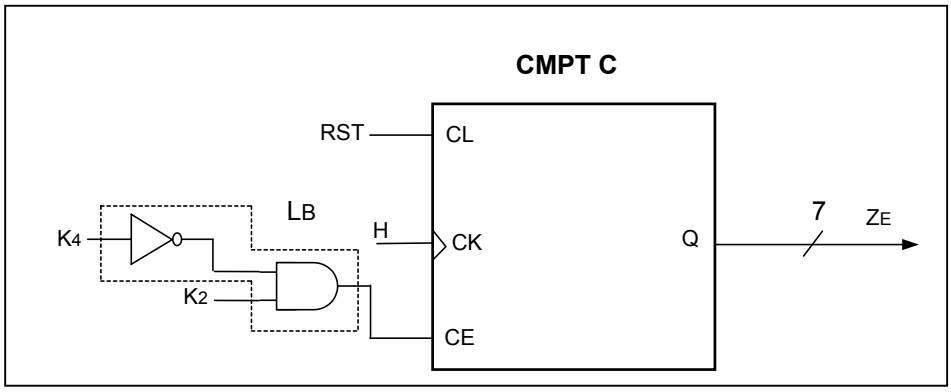

Figure 4.9 : Circuit d'adressage de la mémoire résultat en phase d'expédition.

4.3.1.5. Circuit de sélection d'adresses

Comme il est indiqué sur la figure 4.10, ce circuit est constitué d'un multiplexeur (MUX) commandé par le signal K_2. Il permet la sélection de l'adresse Z_T quand $K_2 = 0$ et la sélection de l'adresse Z_E quand $K_2 = 1$.

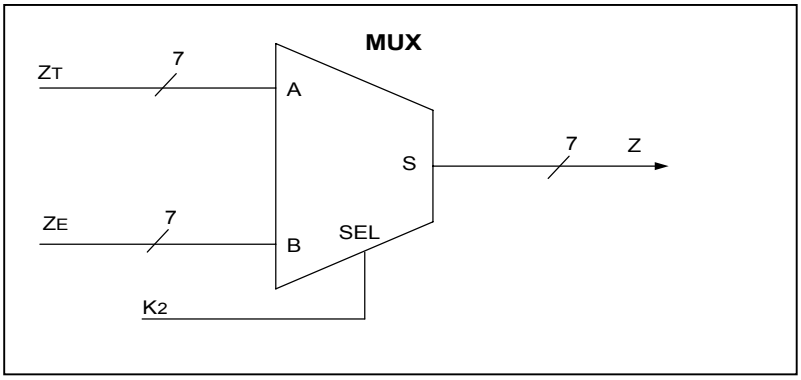

Figure 4.10 : Circuit de sélection d'adresses.

4.3.2. Module de multiplication

Comme il est indiqué sur la figure 4.11, ce module est constitué de quatre parties:

- La première partie est la mémoire MA dans laquelle est rangé le multiplicande A. Elle reçoit en entrée :
  - L'horloge H_2 permettant à chaque front montant de présenter à l'entrée A du multiplieur MUL un mot A_i qui se trouve dans la mémoire MA à l'adresse X.
  - La commande L_C permettant d'activer l'horloge H_2 après 346 cycles de clk.
- La seconde partie est la mémoire MB dans laquelle est rangé le multiplieur B. Elle reçoit en entrée :
  - L'horloge H_2 permettant à chaque front montant de présenter à l'entrée B du multiplieur MUL un mot B_j qui se trouve dans la mémoire MB à l'adresse Y.
  - La commande L_C permettant d'activer l'horloge H_2 après 346 cycles de clk.
- La troisième partie est le multiplieur MUL permettant d'effectuer la multiplication des deux mots A_i et B_j présentés en entrées, en fournissant en sortie le produit $A_i B_j$.
- La quatrième partie est les registres latch L_1 et L_2 permettant de mémoriser respectivement la partie inférieure (P_I) composée des 64 bits les moins significatifs de $A_i B_j$ et la partie supérieure (P_S) composée des 64 bits les plus significatifs de $A_i B_j$ à chaque front montant de l'horloge H_3.

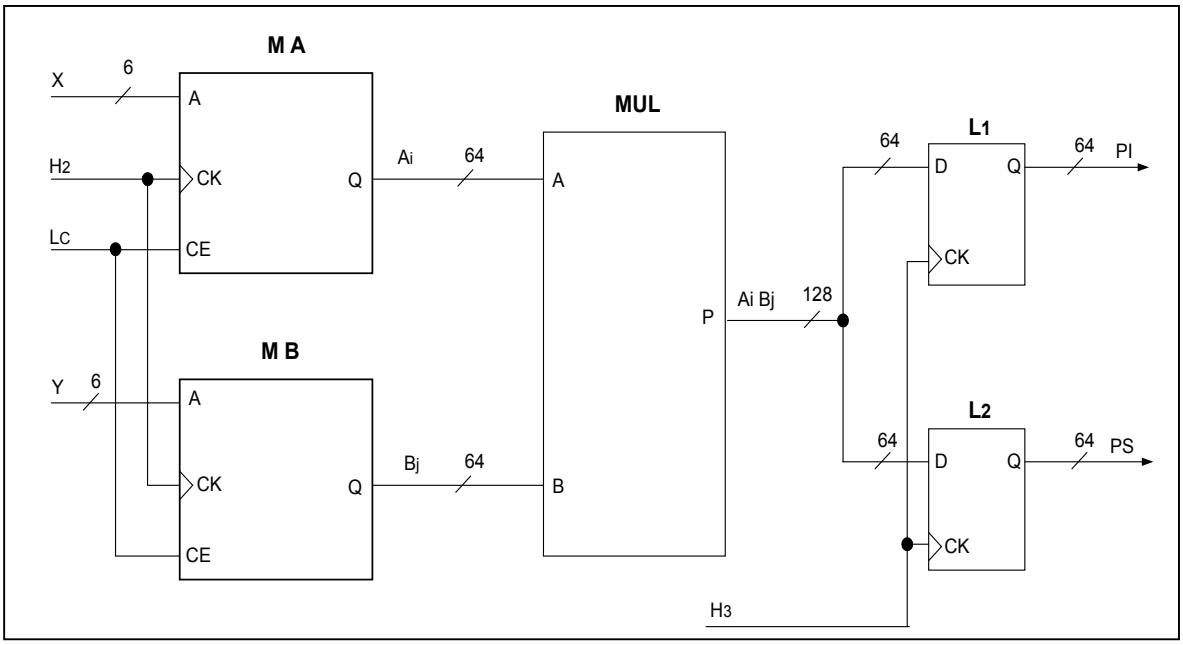

Figure 4.11 : Micro architecture du module de multiplication

4.3.3. Module d'accumulation

Ce module est constitué d'une mémoire et de huit circuits comme il est indiqué sur la figure 4.12.

1. Une mémoire résultat.
2. Un circuit d'addition des poids faibles
3. Un circuit de sélection des retenues.
4. Un circuit d'enregistrement des retenues
5. Un circuit de sélection
6. Un circuit d'addition des poids forts
7. Un circuit d'addition
8. Un circuit de sélection des résultats.
9. Un circuit de transfert des résultats

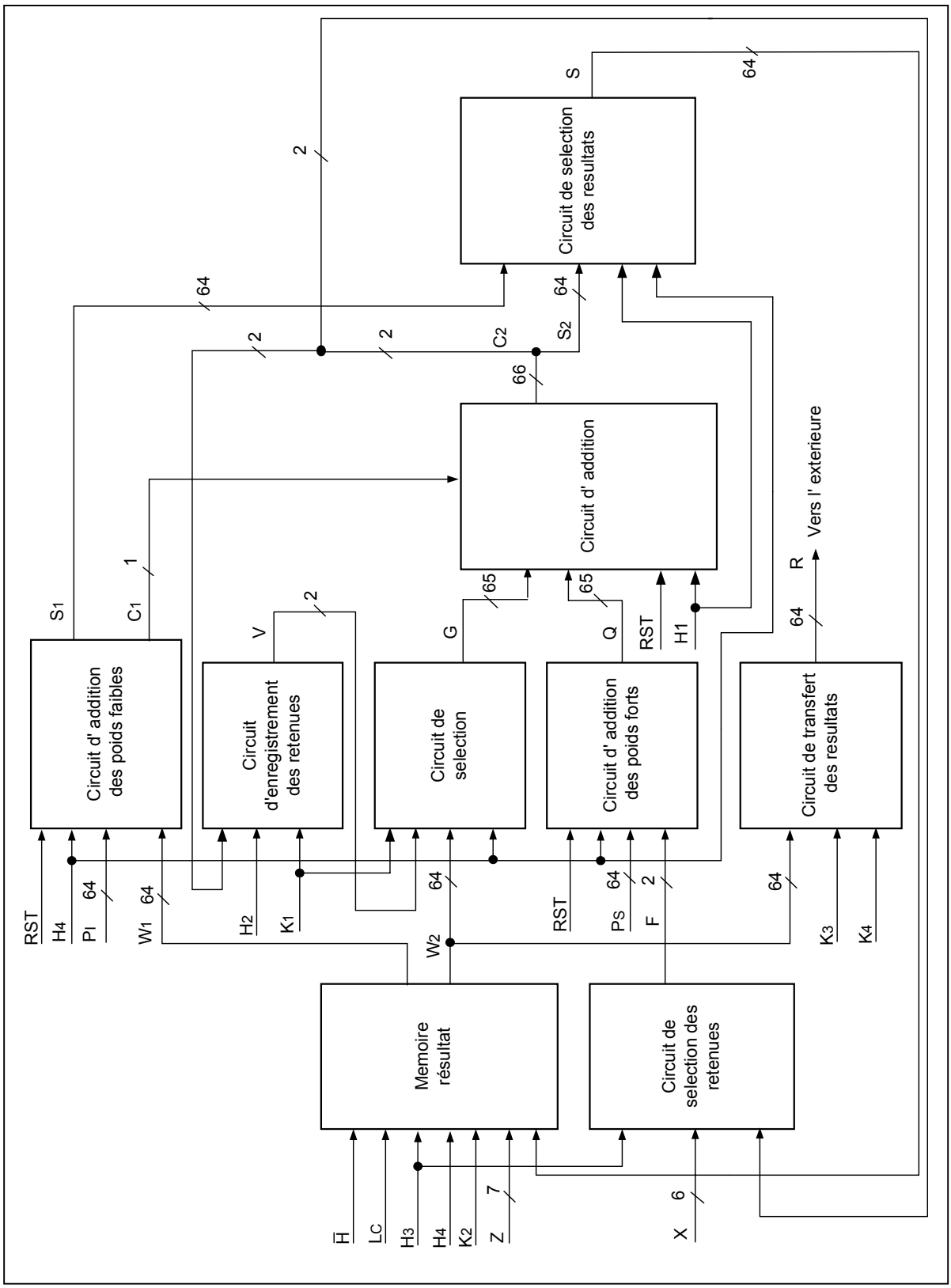

Figure 4.12 : Synoptique de la micro architecture du module d’accumulation.

La mémoire résultat permet pendant chaque séquence du cycle de calcul la restitution des résultats intermédiaires W_1 et W_2 qui se trouvent respectivement aux adresses Z=X+Y et Z=X+Y+1, ensuite le stockage des résultats S_1 et S_2 fournis par le circuit de sélection des résultats aux même adresses (X+Y et X+Y+1). D'autre par, elle permet pendant chaque séquence du cycle d'expédition la restitution du résultat final R qui se trouve à l'adresse Z= Z_E.

Le circuit de sélection des retenues permet d'amener sur la sortie F les deux bits de retenue 00 lors de la multiplication du mot de poids le plus faible du multiplicande (A_0) par le jième mot du multiplieur (B_j), et d'amener sur la même sortie (F) la retenue C_2 fourni par le circuit de sélection des résultats lors des autres cas de séquences.

Le circuit d'addition des poids faibles effectue l'addition de la partie poids faible du produit P_I fourni par le module de multiplication et le résultat intermédiaire W_1 qui se trouve dans la mémoire résultat à l'adresse Z=X+Y, il génère en sortie la somme S_1 et la retenue C_1.

Le circuit d'enregistrement des retenues permet de mettre en mémoire la retenue C_2 fournie par le circuit d'addition lors de la multiplication du jième mot du multiplieur (B_j) par le mot de poids le plus fort du multiplicande (A_{n-1}) pour pouvoir l'exploiter, lors d'une autre séquence de calcul.

Le circuit de sélection permet d'amener sur la sortie G la retenue V fourni par le circuit d'enregistrement des retenues lors de la multiplication du mot de poids le plus fort du multiplicande (A_{n-1}) par le jième mot du multiplieur (B_j), et d'amener sur la même sortie (G) le résultat intermédiaire W_2 qui se trouve dans la mémoire résultat à l'adresse Z=X+Y+1 lors des autres cas de séquences.

Le circuit d'addition des poids forts effectue l'addition de la partie poids fort du produit P_S fourni par le module de multiplication et la retenue F fourni par le circuit de sélection des retenues.

Le circuit d'addition effectue l'addition des deux résultats G et Q issus respectivement des circuits de sélection et d'addition des poids forts, il génère en sortie la somme S_2 et la retenue C_2.

Le circuit de sélection des résultats permet d'amener sur la sortie S à la première époque le résultat S_1 pour qu'il soit rangé dans la mémoire résultat à l'adresse Z=X+Y, et à la seconde époque le résultat S_2 pour qu'il soit rangé dans la mémoire résultat à l'adresse Z=X+Y+1.

Le circuit de transfert des résultats permet d'amener vers l'extérieure du multiplieur à précision variable le résultat final R qui se trouve dans la mémoire résultat à l'adresse $Z=Z_E$ au moment ou le cycle de calcul est terminé.

Dans ce qui suit, nous allons présenter la micro architecture de chaque circuit constituant le module d'accumulation.

4.3.3.1. Mémoire résultat

Comme il est indiqué sur la figure 4.13, cette mémoire reçoit en entrée :

- L'horloge $\overline{H}$ permettant à chaque front montant l'envoi ou la réception d'un mot en mémoire MR.
- La commande L_C permettant d'activer l'horloge $\overline{H}$ après 346 cycles de clk.
- Le signal T généré par le circuit logique (LC) permettant à l'état bas la mise en lecture de MR et à l'état haut la mise en écriture.

Le registre latch (L) permet de mémoriser au front montant de l'horloge H_3 le résultat intermédiaire W_1 lu en mémoire MR à l'adresse Z=X+Y.

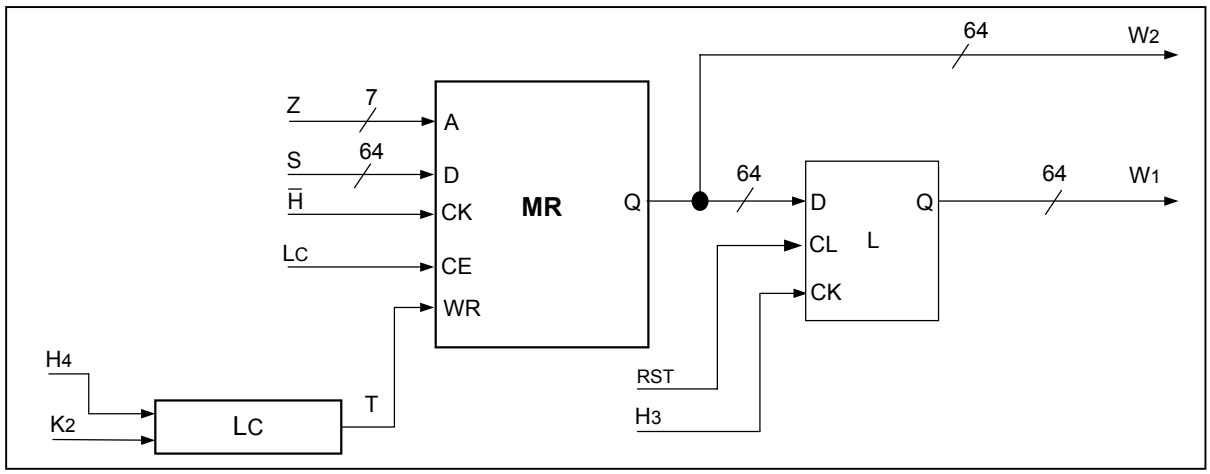

Figure 4.13 : Synoptique de la mémoire résultat.

La table de vérité présentée sur le tableau 4.4 résume le fonctionnement du circuit LC.

K_2	H_4	T	Etat de MR
0	0	0	Lecture
0	1	1	Ecriture
1	0	0	Lecture
1	1	0	Lecture

Tableau 4.4 : Etats du signal de sortie du circuit LC.

Ceux-ci nous permettent de déduire l'expression suivante : $T = H_4 \times \overline{K_2}$

Nous avons traduit cette expression à une circuiterie représenté sur la figure 4.14.

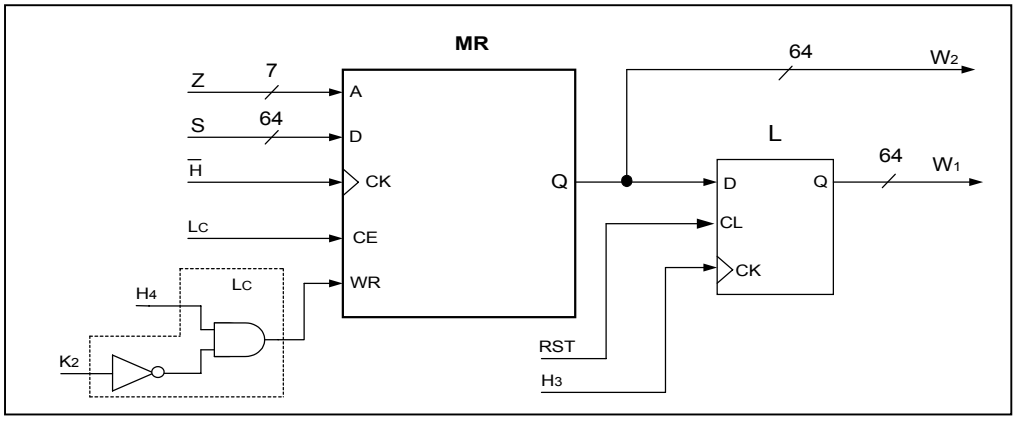

Figure 4.14 : Mémoire résultat

4.3.3.2. Circuit d'addition des poids faibles

Comme il est indiqué sur la figure 4.15, ce circuit est constitué de deux parties :

- La première partie est l'additionneur (ADD) permettant d'effectuer l'addition de la partie poids faible du produit (P_l) fourni par le module de multiplication et le résultat intermédiaire (W_1) lu en mémoire de résultat à l'adresse (X+Y). Il génère en sortie la somme S_1 et la retenu C_1.

- La seconde partie est les deux registres latch L_1 et L_2 permettant de mémoriser respectivement la somme S_1 et la retenu C_1 au front montant de l'horloge H_4.

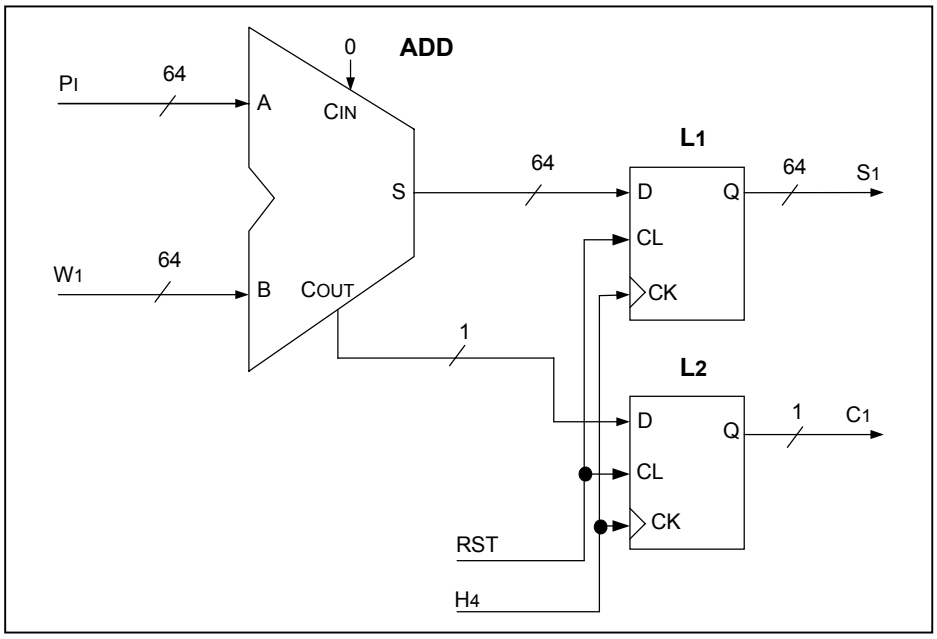

Figure 4.15 : Circuit d'addition des poids faibles.

4.3.3.3. Circuit de sélection des retenues

Comme il est indiqué sur la figure 4.16, ce circuit est constitué de deux parties :

- La première partie est un multiplexeur (MUX) commandé par la sortie T de la porte (OR) qui reçoit en entrée l'adresse X. Si X=0, MUX sélectionne la valeur 00 comme retenue, sinon il sélectionne la retenue d'une séquence précédente C_2 fourni par le circuit d'addition.
- La seconde partie est un registre latch (L) permettant de mémoriser la retenue en sortie de MUX au front montant de l'horloge H_3.

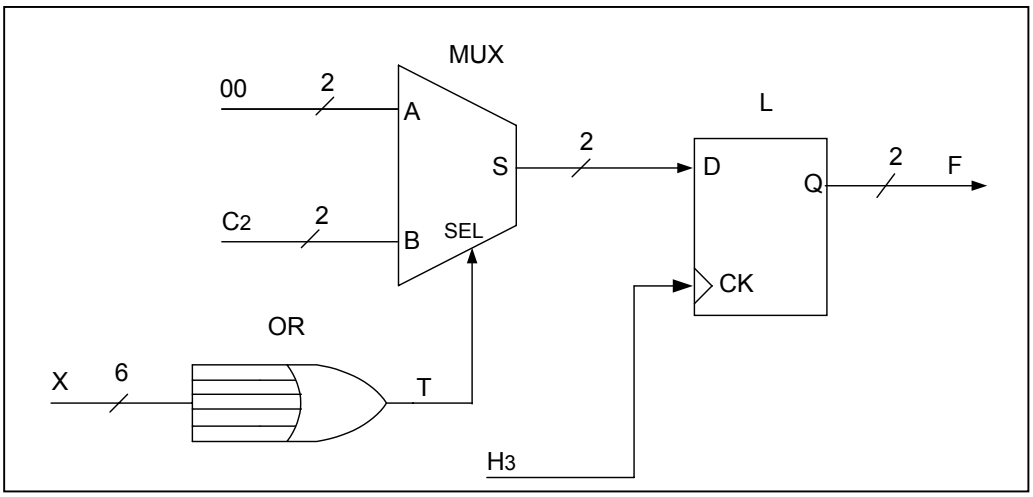

Figure 4.16 : Circuit de sélection des retenues.

4.3.3.4. Circuit d'addition des poids forts

Comme il est indiqué sur la figure 4.17, ce circuit est constitué de deux parties :

- La première partie est l'additionneur (ADD) permettant d'effectuer l'addition de la partie poids fort du produit P_S fourni par le module de multiplication et la retenue F fourni par le circuit de sélection des retenues. Il génère en sortie le résultat Q.
- La seconde partie est le registre (L) permettant de mémoriser le résultat Q en sortie de l'additionneur ADD au front montant de l'horloge H_4.

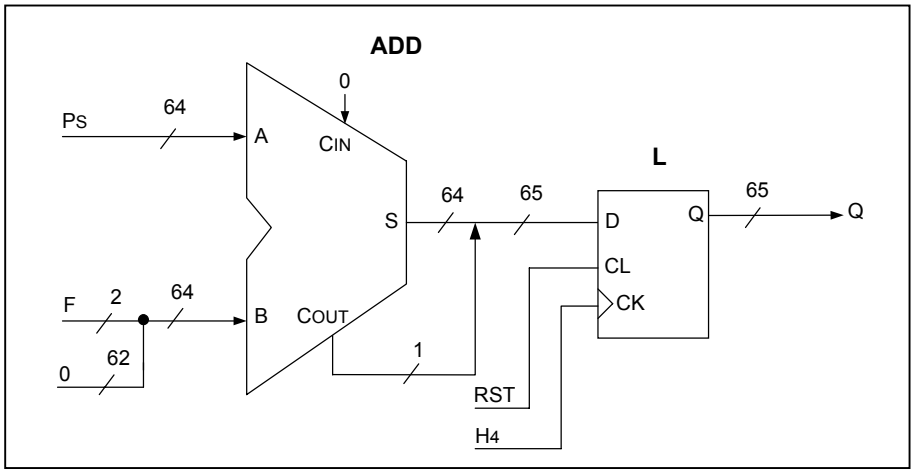

Figure 4.17 : Circuit d'addition des poids forts.

4.3.3.5. Circuit de sélection

Comme il est indiqué sur la figure 4.18, ce circuit est constitué de deux parties :

- La première partie est le multiplexeur (MUX) commandé par le signal K_1. Si K_1=0, MUX sélectionne le résultat intermédiaire W_2 qui se trouve dans la mémoire résultat à l'adresse X+Y, sinon il sélectionne la retenue V fourni par le circuit d'enregistrement des retenues.
- La seconde partie est le registre latch (L) permettant de mémoriser le résultat sélectionné en sortie de MUX au front montant de l'horloge H_4.

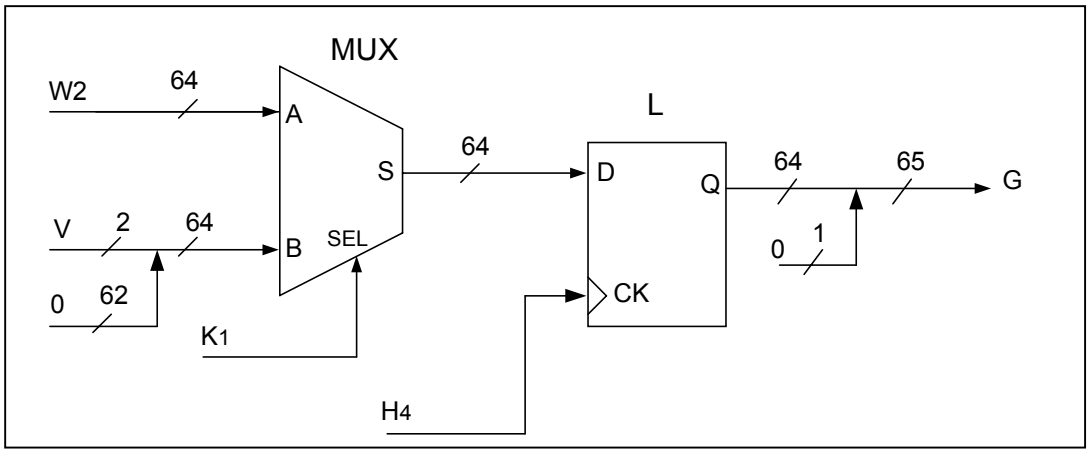

Figure 4.18 : Circuit de sélection.

4.3.3.6. Circuit d'addition

Comme il est indiqué sur la figure 4.19, ce circuit est constitué de deux parties :

- La première partie est l'additionneur (ADD) permettant d'effectuer l'addition du résultat fourni par le circuit d'addition des poids forts (Q) et le résultat fourni par le circuit de sélection (G). Il génère en sortie la somme S_2 et la retenue C_2.
- La seconde partie est le registre latch L permettant de mémoriser la somme et la retenue à la sortie de ADD au front montant de l'horloge H_1.

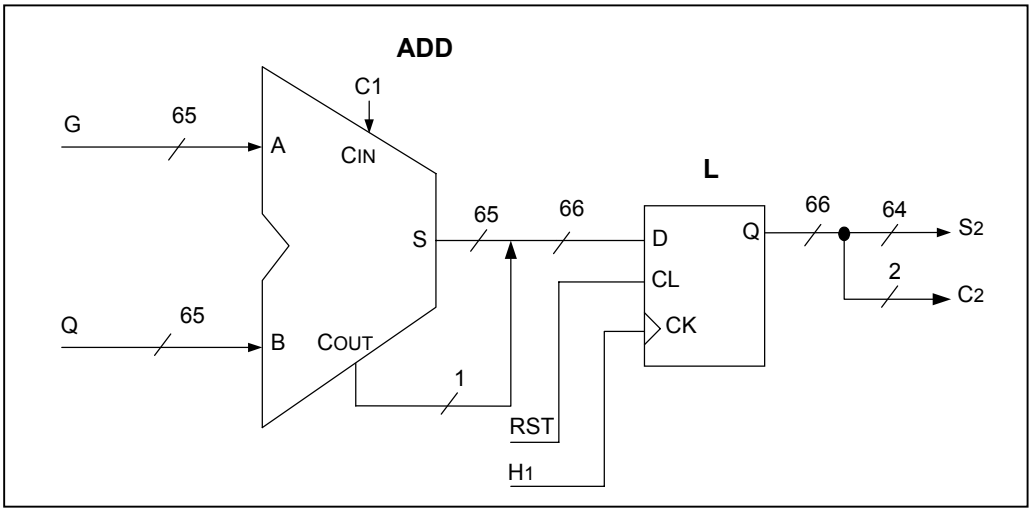

Figure 4.19 : Circuit d'addition.

4.3.3.7. Circuit d'enregistrement des retenues

Comme il est indiqué sur la figure 4.20, ce circuit est constitué d'un registre latch (L). Il reçoit en entrée :

- L'horloge H_2 permettant à chaque front montant de présenté la retenue fournie par le circuit d'addition (C_2) en sortie V du registre L.
- La commande K_1 permettant au niveau haut d'activé l'horloge H_2.

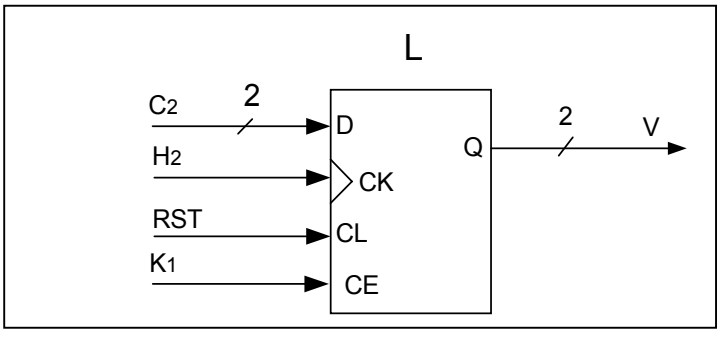

Figure 4.20 : Circuit d'enregistrement des retenues.

4.3.3.8. Circuit de sélection des résultats

Comme il est illustré sur la figure 4.21, ce circuit est constitué d'un multiplexeur (MUX) commandé par le signal F généré par le circuit logique (L_D). Il permet la sélection de la somme (S_1) fournie par le circuit d'addition de la partie poids faible du produit quand F = 0 et la somme (S_2) fournie par le circuit d'addition quand F = 1.

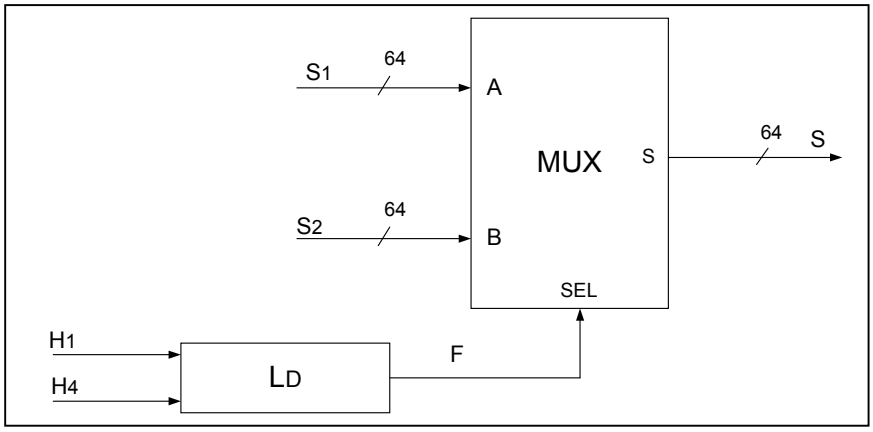

Figure 4.21 : Synoptique du circuit de sélection des résultats

La table de vérité présentée sur le tableau 4.5 résume le fonctionnement du circuit L_D.

H_4	H_1	F	Resultat selectionné (S)
0	0	0	S_1
0	1	0	S_1
1	0	0	S_1
1	1	1	S_2

Tableau 4.5 : Etats du signal de sortie du circuit L_D.

Ceux-ci nous permettent de déduire l'expression suivante : $F = H_1 \times H_4$.

Nous avons traduit cette expression à une circuiterie représenté sur la figure 4.22.

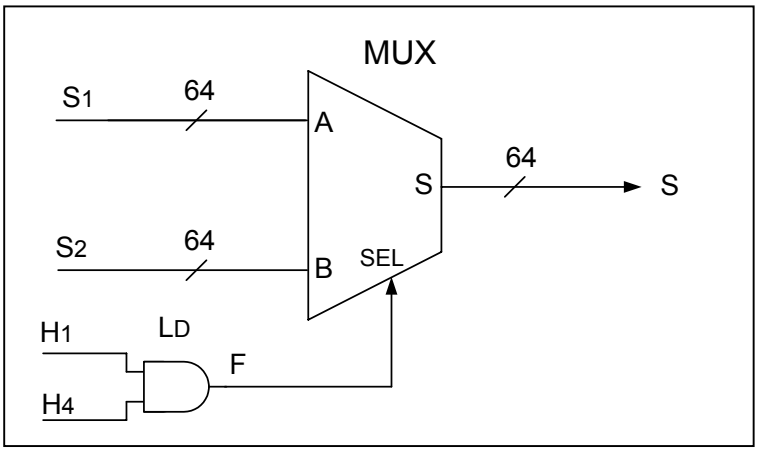

Figure 4.22 : Circuit de sélection des résultats.

.

4.3.3.9. Circuit de transfert des résultats

Comme il est indiqué sur la figure 4.23, ce circuit est constitué d'une porte à trois états (TRS) permettant d'autorise le résultat final R qui se trouvent dans la mémoire MR vers l' extérieur au moment ou le cycle de calcul est terminé. Cette porte est commandée par le signal T qui est généré par le circuit logique (L_E).

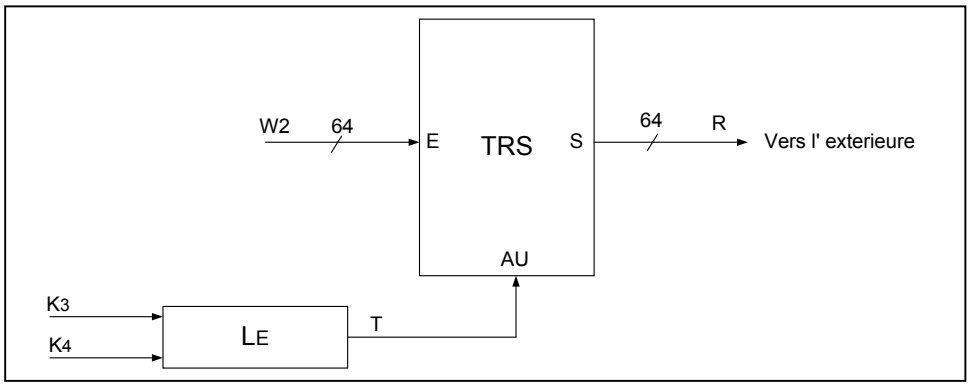

Figure 4.23 : Synoptique du circuit de transfert des résultats.

La table de vérité présentée sur le tableau 4.6 résume le fonctionnement du circuit L_E.

K_4	K_3	T	Résultat (R)
0	0	0	Non autorisé
0	1	1	Autorisé
1	0	0	Non autorisé
1	1	0	Non autorisé

Tableau 4.6 : Etats du signal de sortie du circuit L_E.

Ceux-ci nous permettent de déduire l'expression suivante : $T = K_3 \times \overline{K_4}$.

Nous avons traduit cette expression à une circuiterie représenté sur la figure 4.24.

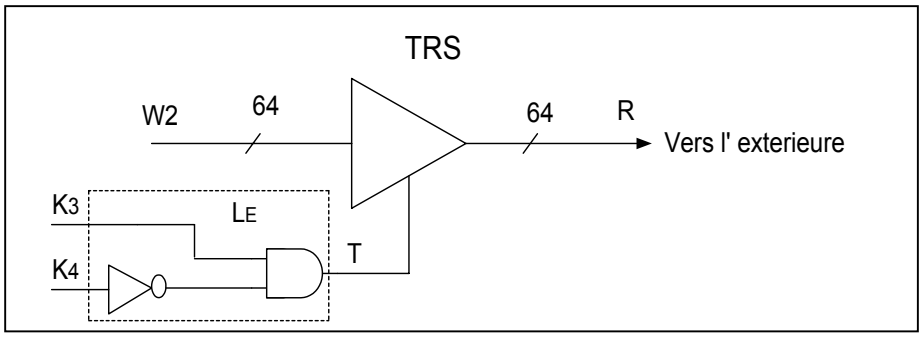

Figure 4.24 : Circuit de transfert des résultats.

4.3.4. Module générateur des signaux d'horloge

Comme il est indiqué sur la figure 4.25, ce module est constitué de trois circuits :

- Le circuit DLL1 destiné à fournir les trois signaux H, $\overline{H}$ et H_{D2} en effectuant un décalage de 0 et 90 degrés et une division par 2 de la fréquence de l'horloge en entrée clk.
- Le circuit DLL2 destiné à fournir l'horloge H_{D4} en effectuant une division par 2 de la fréquence de l'horloge en entrée H_{D2}.
- Le circuit DLL3 destiné à fournir les quatre signaux H_1, H_2, H_3 et H_4 en effectuant des décalages de phase de 0, 90, 180 et 270 degrés de l'horloge en entrée H_{D4}.

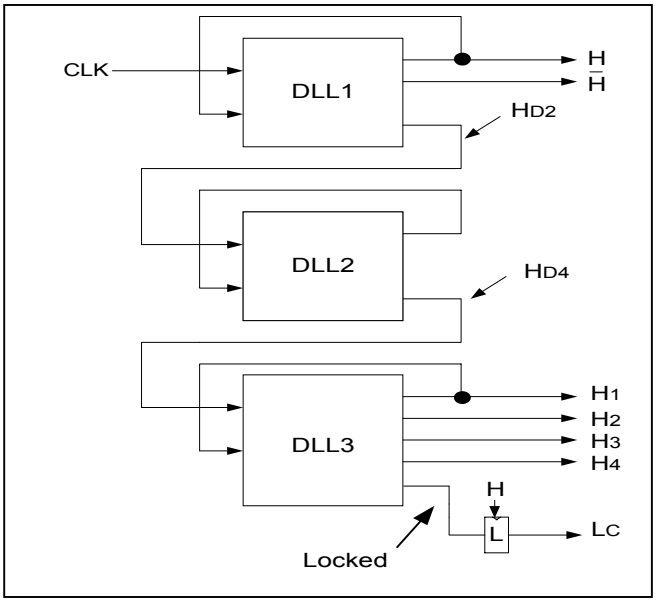

Figure 4.25 : Micro architecture du module générateur des signaux d'horloge.

La forme des signaux générés par ce module est illustrée par les chronogrammes de la figure 4.26.

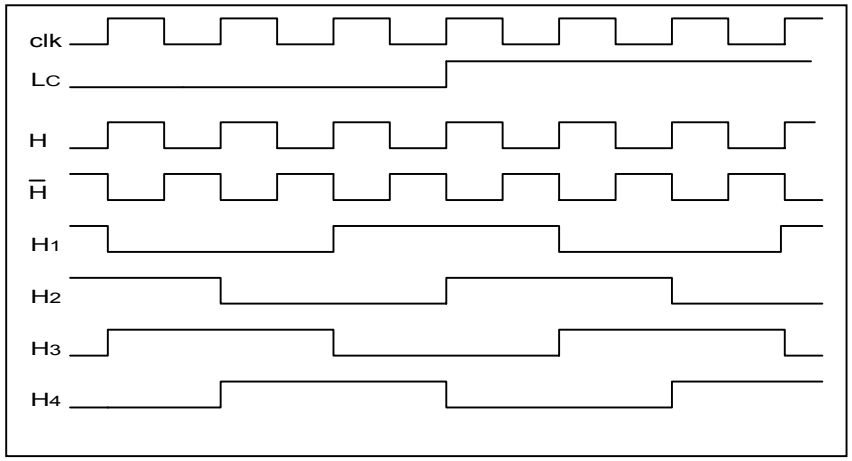

Figure 4.26 : Allure des signaux de sortie du module générateur des signaux d'horloge.

4.3.5. Module générateur des signaux de commande

Ce module est constitué de trois circuits, comme il est indiqué sur la figure 4.27 :

- Un circuit générateur du signal fin étape de calcul.
- Un circuit générateur des signaux fin cycle de calcul et début cycle d'expédition.
- Un circuit générateur du signal fin cycle d'expédition.

Les signaux délivrés par ces circuits sont :

- Le signal de commande K_1 qui indique la fin d'une étape de calcul
- Le signal de commande K_2 qui indique la fin du cycle de calcul
- Le signal de commande K_3 qui indique le début du cycle d'expédition
- Le signal de commande K_4 qui indique la fin du cycle d'expédition

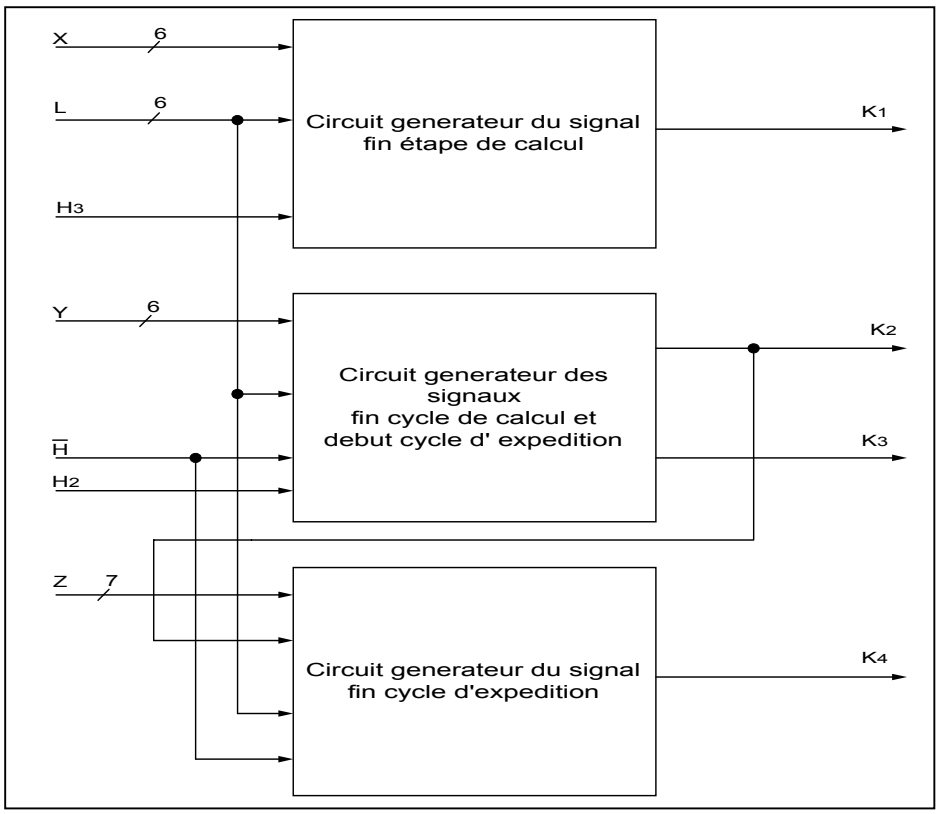

Figure 4.27 : Synoptique de la micro architecture du
module générateur des signaux de commande.

4.3.5.1. Circuit générateur du signal fin étape de calcul

Comme il est indiqué sur la figure 4.28, ce circuit est constitué d'un comparateur (CMP) qui test l'égalité entre l'adresse générée par le module d'adressage (X) et l'adresse extrême de la mémoire multiplicande (L). Sa sortie K_1 passe à l'état haut quand X= L et elle est mémorisée au front montant de l'horloge H_3.

.

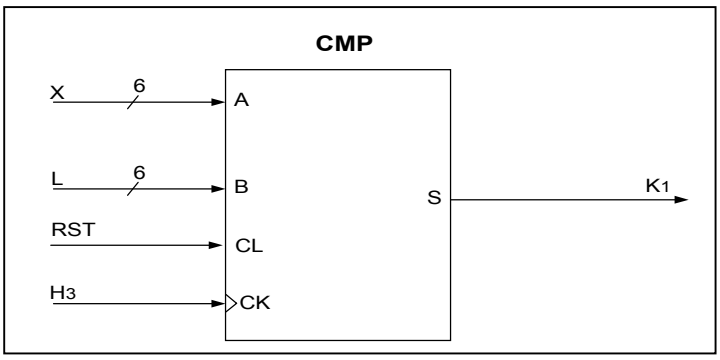

Figure 4.28 : Circuit générateur du signal fin étape de calcul.

4.3.5.2. Circuit générateur des signaux fin cycle de calcul et début cycle d'expédition

Comme il est indiqué sur la figure 4.29, ce circuit est constitué de trois parties :

- La première partie est un comparateur (CMP) qui test l'égalité entre l'adresse générée par le module d'adressage (Y) et l'adresse extrême de la mémoire multiplieur (L). Sa sortie S passe à l'état haut quand Y= L et elle est mémorisée au front montant de l'horloge H_2.
- La seconde partie est une bascule (L_1) qui délivré 1 en sortie K_2 au front montant du signal S.
- La troisième partie est une bascule (L_2) activé par $\overline{H}$ reçoit en entrée le signal K_2 et délivré en sortie le signal K_3 qui passe à 1 dans une demi-période plus tard.

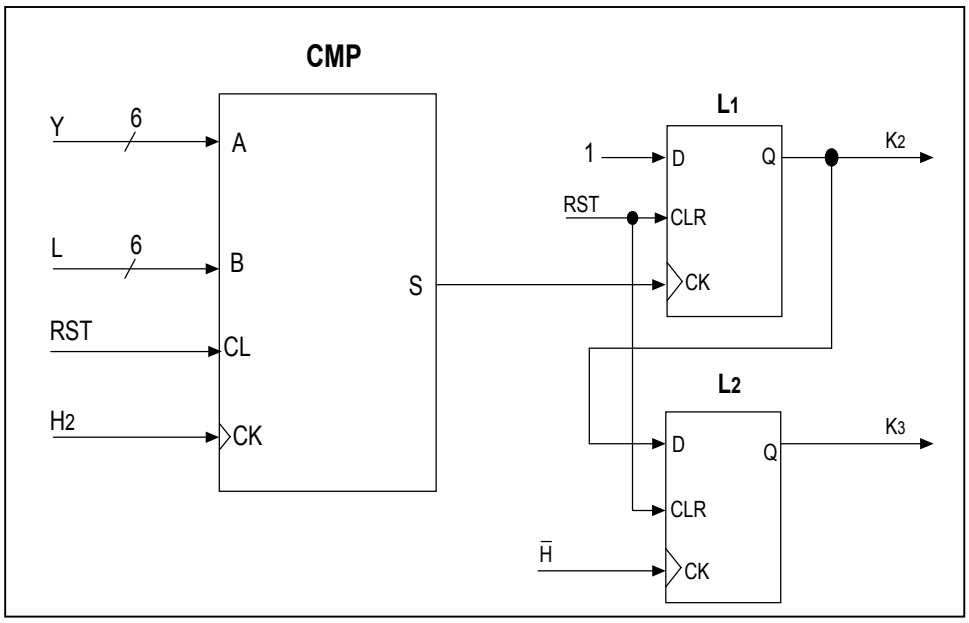

Figure 4.29 : Circuit générateur des signaux fin cycle de calcul et début cycle d' expédition.

4.3.5.3. Circuit générateur du signal fin cycle d'expédition

Comme il est indiqué sur la figure 4.30, ce circuit est constitué de quatre parties :

- La première partie est un additionneur (ADD) qui fourni l'adresse extrême de la mémoire résultat (2L+1).
- La seconde partie est un comparateur (CMP) qui test l'égalité entre l'adresse générée par le module d'adressage (Z) et l'adresse extrême de la mémoire résultat (2L+1). Sa sortie S passe à l'état haut quand Z= 2L+1 et elle est mémorisée au front montant de l'horloge $\overline{H}$. La commande K_2 permet de désactivé $\overline{H}$ pendant le cycle de traitement.
- La seconde partie est une bascule (L_1) qui délivré 1 en sortie Q au front montant du signal S.
- La troisième partie est une bascule (L_2) activé par $\overline{H}$ reçoit en entrée la sortie de L_1 qui vient de passer à 1, sa sortie K_4 passe à 1 dans une demi-période plus tard.

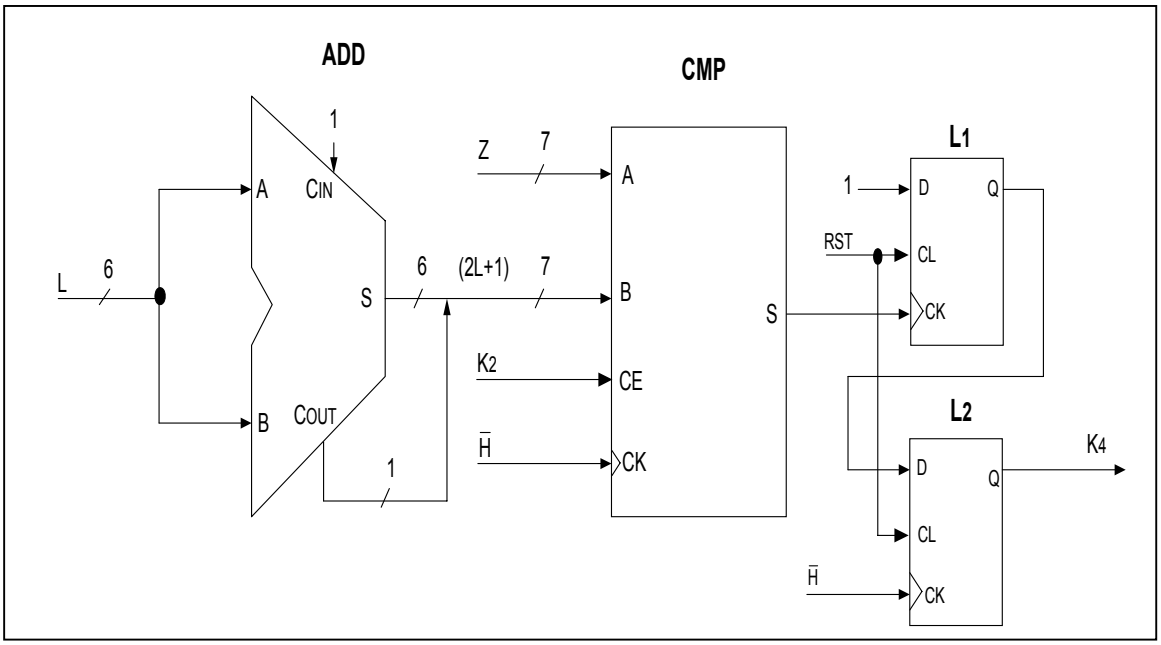

Figure 4.30 : Circuit générateur du signal fin cycle d'expédition.

4.4. Déroulement de l'algorithme dans l'architecture

Afin de comprendre le fonctionnement de la partie traitement du multiplieur à précision variable (voir figure 4.31), nous allons présenter l'exemple suivant :

A= (987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321) $_{Hex}$

B= (987654321FEDCBA987654321FEDCBA9876543FEDCBA98765) $_{Hex}$

L= (02) $_{Hex}$

Les deux nombres A et B à multiplier étant rangés respectivement en mémoires M A et M B à partir de l'adresse (00) $_{Hex}$ à l'adresse (02) $_{Hex}$, la multiplication de A par B s'effectue en accomplissant les itérations suivantes :

*Première itération

- Au top d'horloge H_1
  - ✓ CMPT A et CMPT B étant initialisés à zéro leurs contenus (00) $_{Hex}$ est envoyé vers les mémoires MA et MB.

- Au top d'horloge H_2
  - ✓ Les données (987654FEDCBA4321) $_{Hex}$ et (76543FEDCBA98765) $_{Hex}$ lues respectivement en mémoires MA et MB sont envoyés vers le multiplieur MUL.

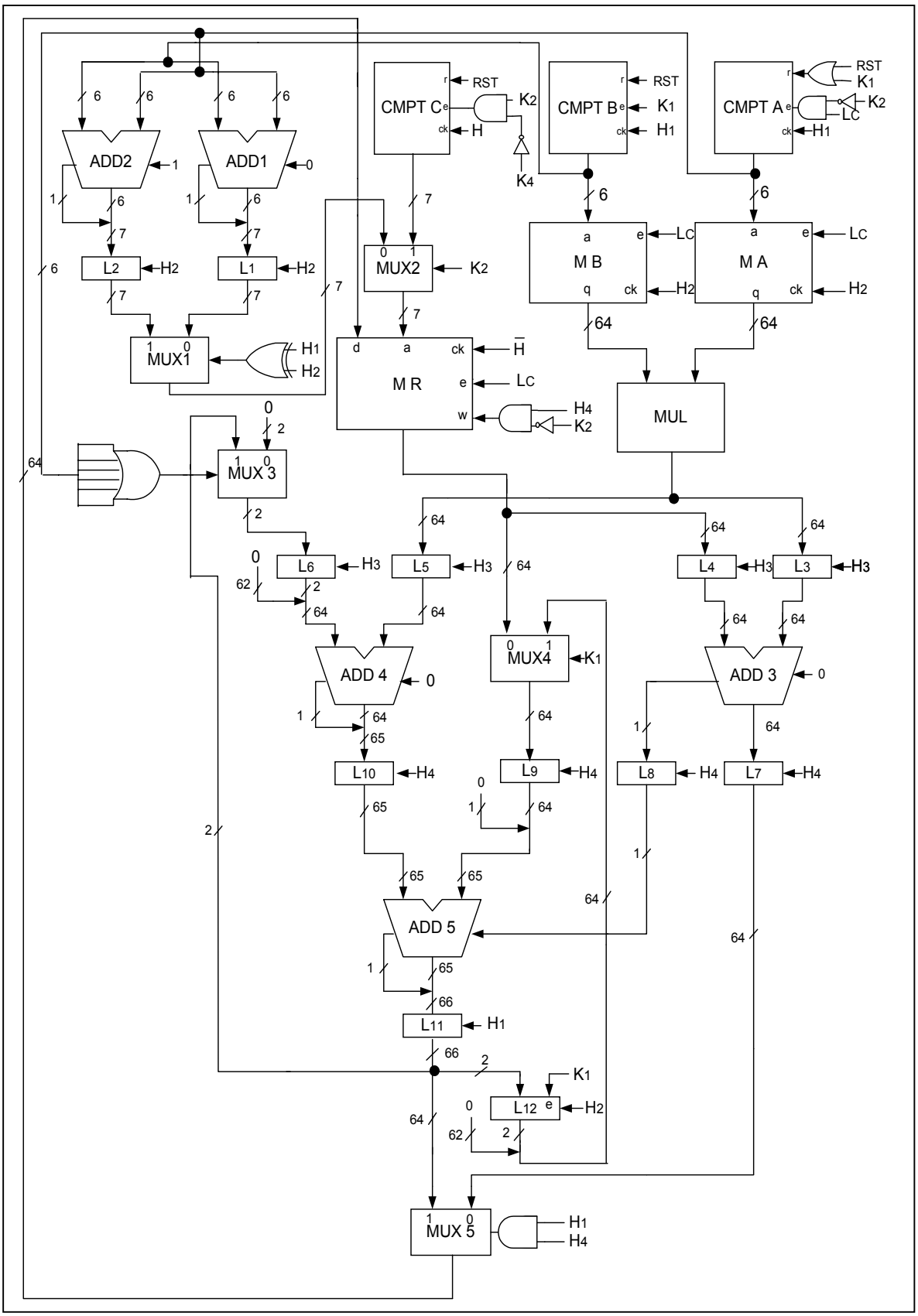

Figure 4.31 : Micro architecture de la partie opérative (traitement) du multiplieur à précision variable.

- ✓ L_1 et L_2 reçoivent $(00)_{Hex}$ et $(01)_{Hex}$ comme adresses à mémoriser et à envoyer vers la mémoire MR.

- Au top d'horloge H_3
  - ✓ L_3 et L_4 reçoivent $(2E929DB1CAABE305)_{Hex}$ et $(0000000000000000)_{Hex}$ comme données à mémoriser et à envoyer vers l'additionneur ADD3.
  - ✓ L_5 et L_6 reçoivent $(4678B8141B207C7A)_{Hex}$ et $(0000000000000000)_{Hex}$ comme données à mémoriser et à envoyer vers l'additionneur ADD4.
- Au top d'horloge H_4
  - ✓ L_7 reçoit $(2E929DB1CAABE305)_{Hex}$ comme donnée à mémoriser et à envoyer vers la mémoire MR.
  - ✓ L_8 reçoit 0 comme donnée à mémoriser et à envoyer vers l'additionneur ADD5.
  - ✓ L_9 et L_{10} reçoivent $(0000000000000000)_{Hex}$ et $(04678B8141B207C7A)_{Hex}$ comme données à mémoriser et à envoyer vers l'additionneur ADD5.

*Deuxième itération

- Au top d'horloge H_1
  - ✓ L_{11} reçoit $(04678B8141B207C7A)_{Hex}$ comme donnée à mémoriser et à envoyer vers la mémoire MR.
  - ✓ CMPT A est incrémenté et son contenu $(01)_{Hex}$ est envoyé vers la mémoire MA.
  - ✓ CMT B étant initialisé à zéro et son contenu $(00)_{Hex}$ est envoyé vers la mémoire MB
- Au top d'horloge H_2
  - ✓ Les données $(987654FEDCBA8765)_{Hex}$ et $(76543FEDCBA98765)_{Hex}$ lues respectivement en mémoires MA et MB sont envoyés vers le multiplieur MUL.
  - ✓ L_1 et L_2 reçoivent $(01)_{Hex}$ et $(02)_{Hex}$ comme adresses à mémoriser et à envoyer vers la mémoire MR.
- Au top d'horloge H_3
  - ✓ L_3 et L_4 reçoivent $(FDEEC2F4EBAAADD9)_{Hex}$ et $(4678B8141B207C7A)_{Hex}$ comme données à mémoriser et à envoyer vers l'additionneur ADD3.
  - ✓ L_5 et L_6 reçoivent $(4678B8141B207C7A)_{Hex}$ et $(0000000000000000)_{Hex}$ comme données à mémoriser et à envoyer vers l'additionneur ADD4.
- Au top d'horloge H_4
  - ✓ L_7 reçoit $(44677B0906CB2A53)_{Hex}$ comme donnée à mémoriser et à envoyer vers la mémoire MR.

- ✓ L_8 reçoit 1 comme donnée à mémoriser et à envoyer vers l'additionneur ADD5.
- ✓ L_9 et L_{10} reçoivent $(0000000000000000)_{Hex}$ et $(4678B8141B209C07)_{Hex}$ comme données à mémoriser et à envoyer vers l'additionneur ADD5.

*Troisième itération

- Au top d'horloge H_1
  - ✓ L_{11} reçoit $(04678B8141B209C08)_{Hex}$ comme donnée à mémoriser et à envoyer vers la mémoire MR.
  - ✓ CMPT A est incrémenté et son contenu $(02)_{Hex}$ est envoyé vers la mémoire MA.
  - ✓ CMPT B étant initialisé à zéro et son contenu $(00)_{Hex}$ est envoyé vers la mémoire MB.
- Au top d'horloge H_2
  - ✓ Les données $(987654FEDCBACBA9)_{Hex}$ et $(76543FEDCBA98765)_{Hex}$ lues respectivement dans MA et MB sont envoyés vers le multiplieur MUL.
  - ✓ L_1 et L_2 reçoivent $(02)_{Hex}$ et $(03)_{Hex}$ comme adresses à mémoriser et à envoyer vers la mémoire MR.
- Au top d'horloge H_3
  - ✓ L_3 et L_4 reçoivent $(CD4AE83880CA978765)_{Hex}$ et $(4678B8141B209C08)_{Hex}$ comme données à mémoriser et à envoyer vers l'additionneur ADD3.
  - ✓ L_5 et L_6 reçoivent $(4678B8141B20BB95)_{Hex}$ et $(0000000000000000)_{Hex}$ comme données à mémoriser et à envoyer vers l'additionneur ADD4.
- Au top d'horloge H_4
  - ✓ L_7 reçoit $(13C3A04C27C27CA1425)_{Hex}$ comme donnée à mémoriser et à envoyer vers la mémoire MR.
  - ✓ L_8 reçoit 0 comme donnée à mémoriser et à envoyer vers l'additionneur ADD5.
  - ✓ L_9 et L_{10} reçoivent $(0000000000000000)_{Hex}$ et $(4678B8141B209C08)_{Hex}$ comme données à mémoriser et à envoyer vers l'additionneur ADD5.
  - ✓ L_{11} reçoit $(4678B8141B20BB96)_{Hex}$ comme donnée à mémoriser et à envoyer vers la mémoire MR.

A ce stade la première étape de la multiplication A×B parait terminer et on a le début de la seconde étape, ce soit bien le second mot de B a 64 bits qui est adressé, on aura le résultat (0..... $02E929DB1CAABE305)_{Hex}$ dans la mémoire MR qui est utilisé pour cette étape.

Au début de chaque étape le CMPT A est initialisé à zéro et le compteur CMPT B est incrémenté et si la sortie de ces compteurs atteint $(02)_{Hex}$, le cycle de calcul est terminé et on aura

le résultat R= A×B= (5ACCBB272E8CD45D52D5D8862E0D3E24E59C081E31E08666B8D93AB6E02B02B1EC50 AE7A8D884FFEB2E929DB1CAABE305)$_{Hex}$ dans la mémoire MR.

4.5. Conclusion

Dans ce chapitre, nous avons présenté l'architecture du multiplieur à précision variable. Cette architecture est composée essentiellement de cinq modules : module d'adressage, module de multiplication, module d'accumulation, module générateur des signaux d'horloge et module générateur des signaux de commande. Chacun de ces modules est constitué d'un ou de plusieurs circuits, leurs micro-architectures ainsi que leurs principes de fonctionnement ont été présentés en détailles.

Dans le chapitre suivant, nous allons présenter la méthodologie de conception sur circuit FPGA.

CHAPITRE 5

METHODOLOGIE DE CONCEPTION SUR CIRCUIT FPGA

5.1. Introduction

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

5.2. Les FPGAs (Field Programmable Gate Arrays)

FPGA se traduit en français par circuits pré-diffusés programmables. Contrairement aux circuits pré-diffusés conventionnels, les circuits pré-diffusés programmables ne demandent pas de fabrication spéciale en usine, ni de systèmes de développement coûteux. Inventés par la société Xilinx, le FPGA, dans la famille des ASICs, se situe entre les réseaux logiques programmables et les pré diffusés. C'est donc un composant standard combinant la densité et les performances d'un pré diffusé avec la souplesse due à la programmation des PLD. Cette configuration évite le passage chez le fondeur et tous les inconvénients qui en découlent [8].

5.2.1. Description d'un FPGA

Le FPGA fournit le réseau de portes comme architecture, il forme une matrice carrée (N×N) arrangé d'une manière particulière selon les trois éléments de base dont il dispose : un périmètre de blocs d'entrée/sortie (IOB BLOCKS), un corps qui un assemblage de blocs logiques configurables (LOGIC BLOCS) et des matrices d'interconnections programmables (PROGRAMMABLE INTERCONNECT). La figure 5.1 présente l'architecture interne du FPGA [8].

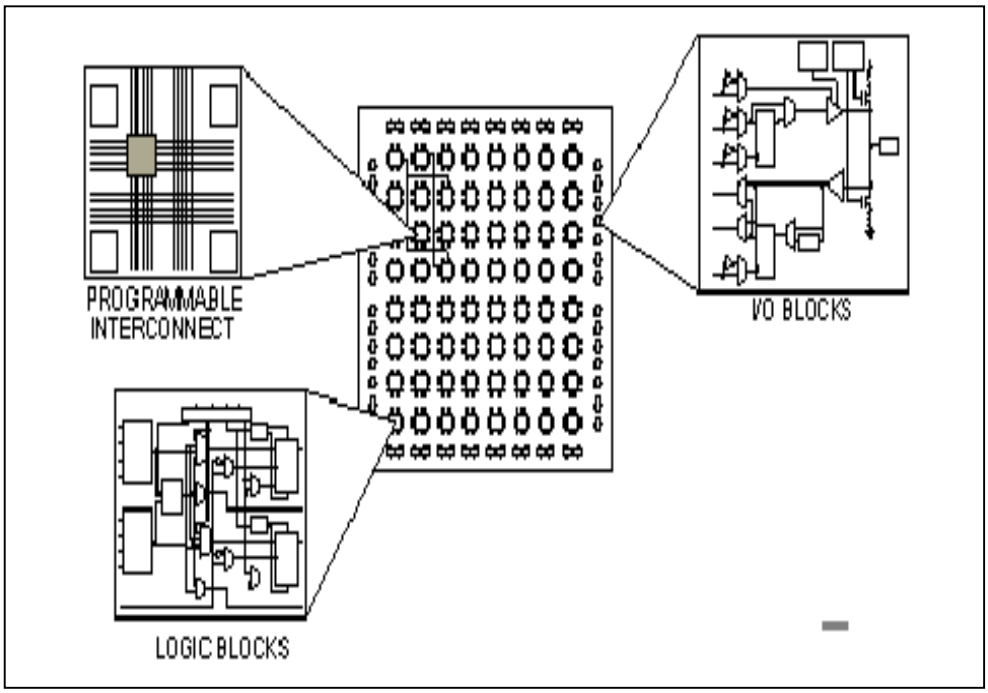

Figure 5.1 : Architecture interne du FPGA.

Chaque bloc est constitué de quelques éléments pour réaliser une tache bien spécifiée [8] :

- Bloc logique configurable (CLB) : c'est le cœur du circuit FPGA, il décrit la fonction qui lui est assigné(voir figure 5.2), il est constituer de :
  - Générateur de fonction combinatoire
  - Bascule flip –flop
  - Multiplexeur de signaux de contrôle
  - Ligne d'horloge

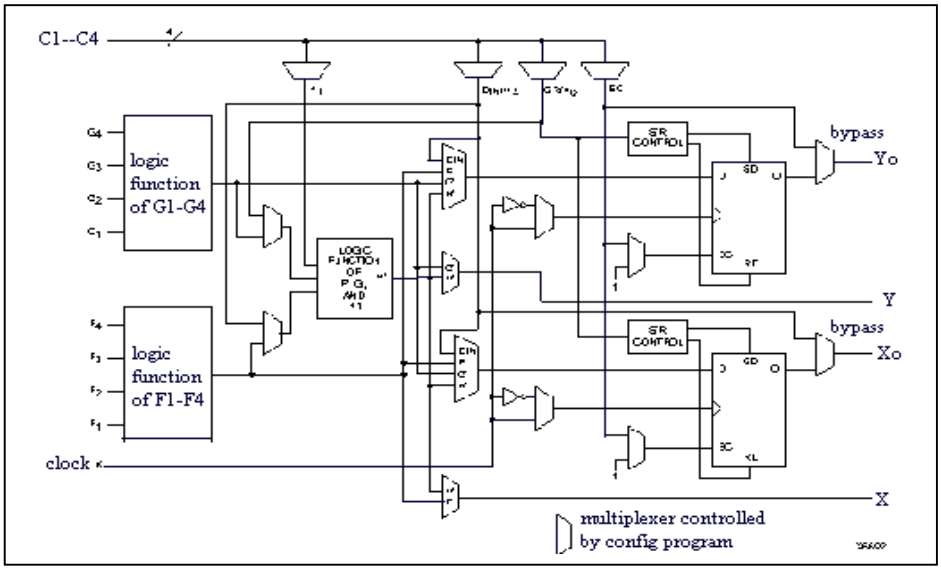

Figure 5.2 : Bloc logique configurable (CLB).

- Bloc d'entrée/sortie (IOB) : c'est le contour du circuit FPGA ou l'interface entre les broches du composant FPGA et la logique interne développée a l'intérieur du circuit, il contrôle une broche du composant, pouvant ainsi la configurer en entrée, en sortie, en bidirectionnel ou en haute impédance(voir figure 5.3).

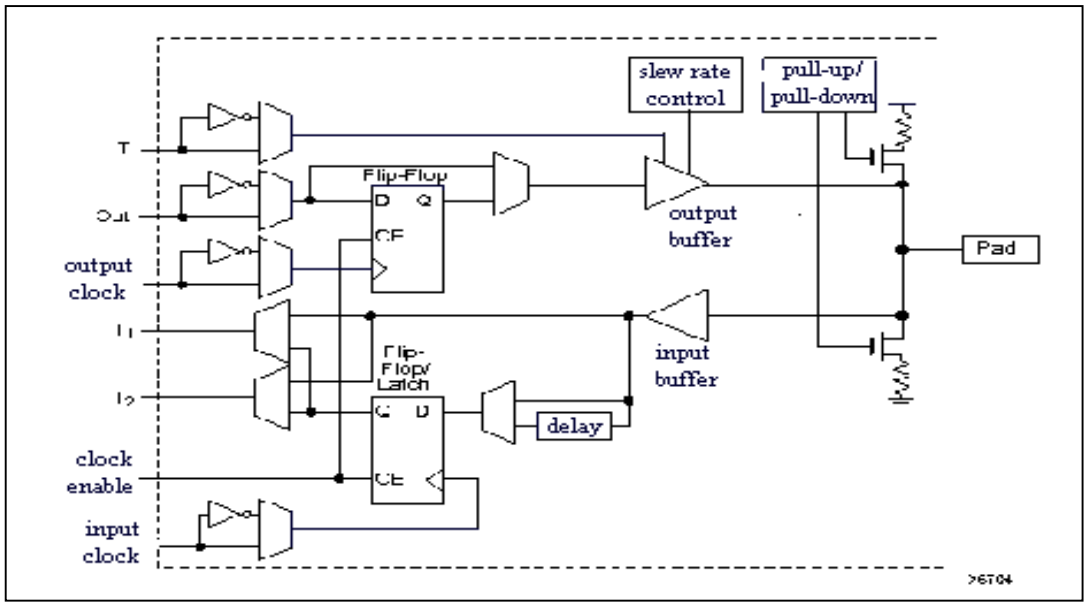

Figure 5.3 : Bloc d'entrée/sortie (IOB)

- Lignes d'interconnexions : c'est un segment de droites parallèles horizontales et perpendiculaires avec des points d'interconnexions programmables pour permettre en application le routage désiré :
  - Le routage CLB associé a chaque ligne et colonne de l'assemblage CLB.
  - Le routage IOB connecte les entrées sorties avec les blocs logiques internes.
  - Le routage général qui affecte les horloges dans tout le FPGA d'une manière parfaite avec un minimum de délais.

Selon la longueur de la destination des liaisons, on distingue trois sortes d'interconnexions :

- Les interconnexions à usage général : elles sont constituées d'une grille de cinq segments métalliques verticaux et quatre segments horizontaux positionnés entre les rangées et les colonnes de CLB et de l'IOB (voir figure 5.4). Des aiguilleurs appelés aussi matrice de commutation sont situés à chaque intersection. Leur rôle est de raccorder les segments entre eux selon diverses configurations, ils assurent ainsi la communication des signaux d'une voie sur l'autre. Ces interconnexions sont utilisées pour relier un CLB à n'importe quel autre.

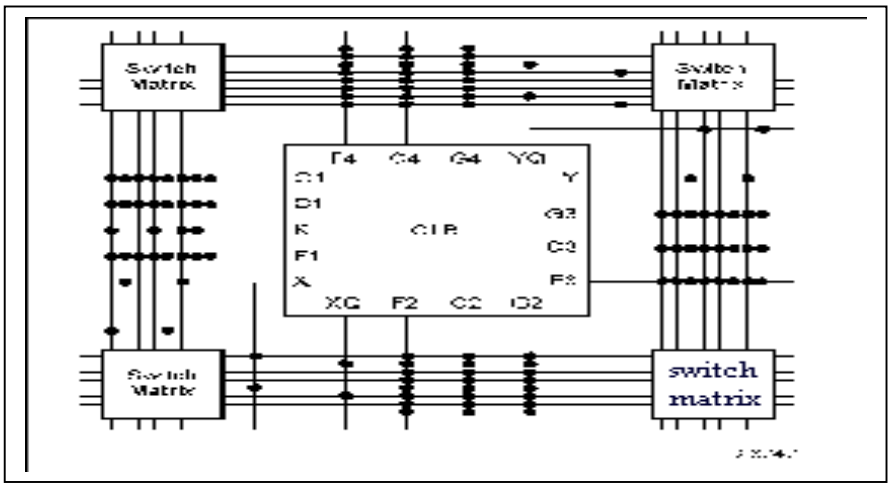

Figure 5.4 : Les interconnexions à usage général.

- Les interconnexions directes : ces interconnexions permettent l'établissement de liaisons entre les CLB et les IOB avec un maximum d'efficacité en terme de vitesse et d'occupation du circuit (voir figure 5.5). De plus, il est possible de connecter directement certaines entrées d'un CLB aux sorties d'un autre.

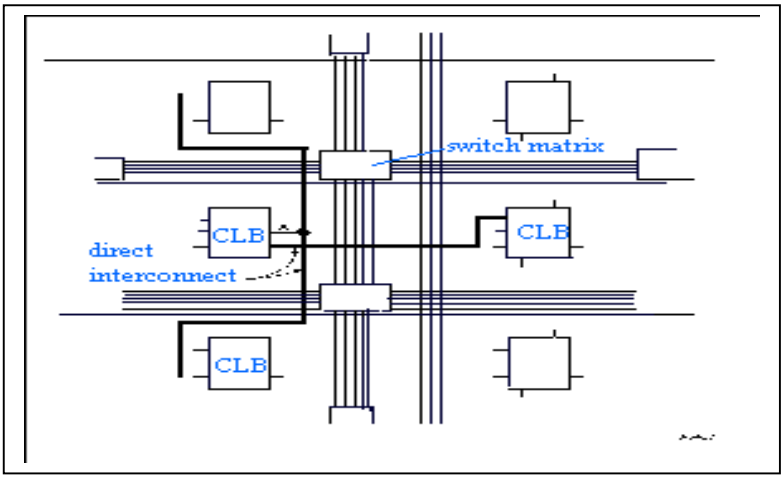

Figure 5.5 : Les interconnexions directes

- Les interconnexions lignes : ce sont de longs segments métallisés parcourant toute la longueur et la largeur du composant (voir figure 5.6), elles permettent éventuellement de transmettre avec un minimum de retard les signaux entre les différents éléments dans le but d'assurer un synchronisme aussi parfait que possible. De plus, ces longues lignes permettent d'éviter la multiplicité des points d'interconnexion.

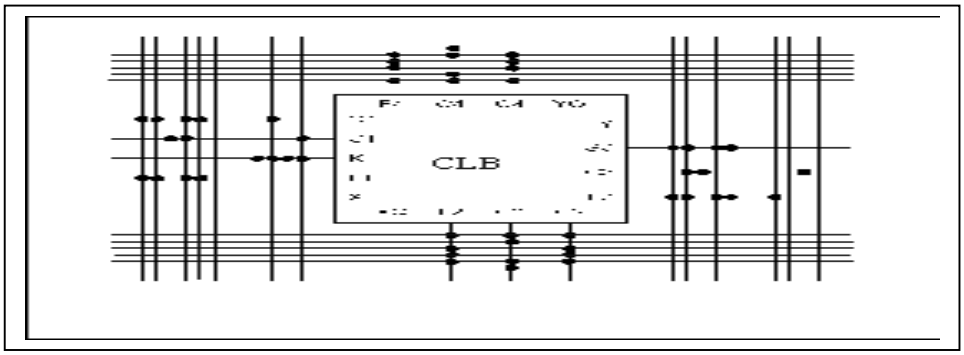

Figure 5.6 : Les interconnexions lignes.

5.2.2. Type d'FPGA

Il existe actuellement plusieurs fabricants de circuits FPGA dont Xilinx et Altera sont les plus connus, les différentes versions des FPGA de Xilinx sont : Spartan-II, Virtex-IIE, Virtex, Virtex-E et Virtex-II. Leurs caractéristiques principales sont :

- Complexités allant de 1500 a plus de 8 millions de portes logiques.
- Faible consommation.
- Grande souplesse d'utilisation des entrées/sorties avec adaptation d'impédance (Virtex-E) et (Virtex-II) et configuration en mode différentielle (Spartan-IIE, Virtex-E et Virtex-II).
- Fonction (mémoire distribuée et blocs de RAM).
- Dispositifs de gestion des horloges(DLL et DCM).
- Multiplieurs câblés.

5.2.2.1 Famille Virtex II

La famille Virtex-II est le premier jeu de plat-forme FPGA, elle a été conçue pour réaliser des conceptions a faible ou grande densité d'intégration et exigent des performances élevés (elle peut et atteindre jusqu'a 10 million de portes logiques). La fréquence de son utilisation peut être portée à 420 MHZ. Cette famille a onze membres (voir tableau 5.1) dans la densité de portes système s'étendant de 40Ko à 8M [9].

Actuellement, cette famille est utilisée dans plusieurs applications telles que la telecommmunication, les réseaux, la vidéo et les applications DSP (Digital Signal Processing) [9].

Virtex-II Device	Maximum System Gates	Logic Cells	Embedded BRAM Memory	Maximum Distributed Memory	18x18 Multiplier Blocks	DCMs	Maximum I/Os
XC2V8000	8M	104,832	3.024 Mbits	1.456 Mbits	168	12	1,108
XC2V6000	6M	76,032	2.592 Mbits	1.056 Mbits	144	12	1,104
XC2V4000	4M	51,840	2.160 Mbits	720 Kbits	120	12	912
XC2V3000	3M	32,256	1.728 Mbits	448 Kbits	96	12	720
XC2V2000	2M	24,192	1.008 Mbits	336 Kbits	56	8	624
XC2V1500	1.5M	17,280	864 Kbits	240 Kbits	48	8	528
XC2V1000	1M	11,520	720 Kbits	160 Kbits	40	8	432
XC2V500	500K	6,912	576 Kbits	96 Kbits	32	8	264
XC2V250	250K	3,456	432 Kbits	48 Kbits	24	8	200
XC2V80	80K	1,152	144 Kbits	16 Kbits	16	4	120
XC2V40	40K	596	72 Kbits	8 Kbits	8	4	88

Tableau 5.1 : Membres de la famille Virtex-II.

Actuellement, cette famille est utilisée dans plusieurs applications telles que la telecommmunication, les réseaux, la vidéo et les applications DSP (Digital Signal Processing) [9].

5.2.2.1.1. Nomenclature d'un circuit Virtex-II

Les circuits $FPGA_S$ de Xilinx sont caractérisés par une nomenclature spécifique qui définie les performances de chaque famille, cette nomenclature est illustrée dans la figure 5.7.

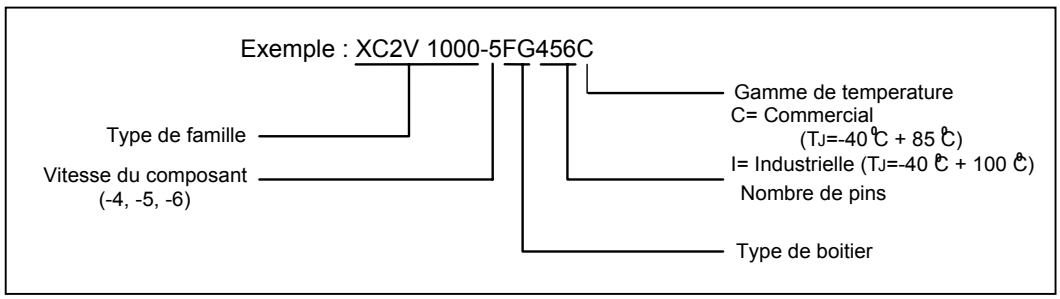

Figure 5.7 : La nomenclature d'un circuit Virtex-II.

Ou :

XC : Xilinx Circuit.

2V : « Device type » : elle représente le type de famille, dans ce cas c'est : Virtex-II.

1000 : c'est le nombre de portes logiques, dans ce cas il s'agit de 1M portes.

5 : « Speed Grade » : c'est la vitesse du composant selon la technologie utilisée dans la fabrication du circuit.

FG : « Package type » : elle représente le type de boîtier.

456 : « Number of pins » : elle indique le nombre de pins dans le circuit.

C : « Temperature Range » : elle donne la gamme de température tolérée [9].

5.2.2.1.2. Architecture d'un circuit Virtex II

L'architecture d'un circuit Virtex-II est montrée sur la figure 5.8, elle est constituée de quatre blocs principaux qui sont :

- Bloc logique configurable (CLB).
- Bloc d'entrée/sortie (IOB)
- Bloc mémoire à 18Kbits (Bloc Select RAM).
- Bloc multiplieur 18bits×18bits (Multiplier).
- Bloc directeur de pendule à lecture digitale (DCM).

La connexion entre ces blocs est assurée d'une manière programmable grâce à des ressources d'interconnections configurables GRM "General Routing Matrix".

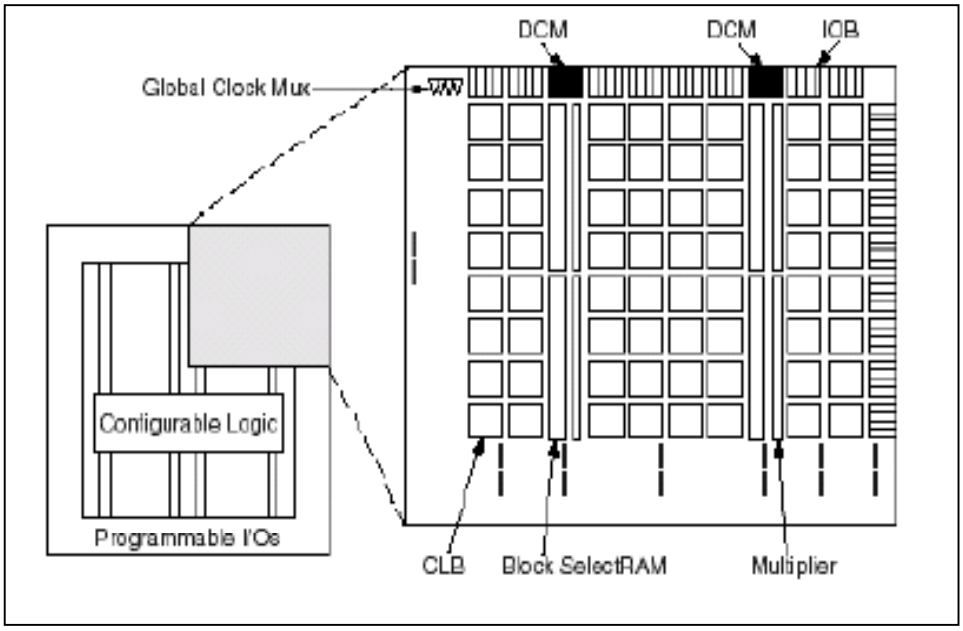

Figure 5.8 : Architecture interne du Virtex-II.

La logique interne configurable inclut quatre éléments majeurs organisés sous une forme régulier et un chemin dédié a la propagation de la retenue [9].

5.2.2.1.2.1. Les CLB_S (Configurable Logic Bloc)

Les CLBs sont implémentés sous une matrice (N×M) sur un circuit FPGA, ils incluent toute la logique nécessaire pour la conception des circuits combinatoires et séquentiels. Chaque CLB est composé de quatre slices(voir figure 5.9) et trois portes à trois états, chaque slice contient [10] :

- Deux générateurs de fonction (F et G).
- Deux éléments de stockage.
- Des portes arithmétiques et logiques.
- Des grands multiplexeurs.
- Une large capacité de fonction.
- Une chaîne en cascade horizontale (porte OU).

Les générateurs de fonction F et G sont configurables comme des LUT_S à quatre entrées, comme des registres a décalage, ou comme des mémoires SRAM.

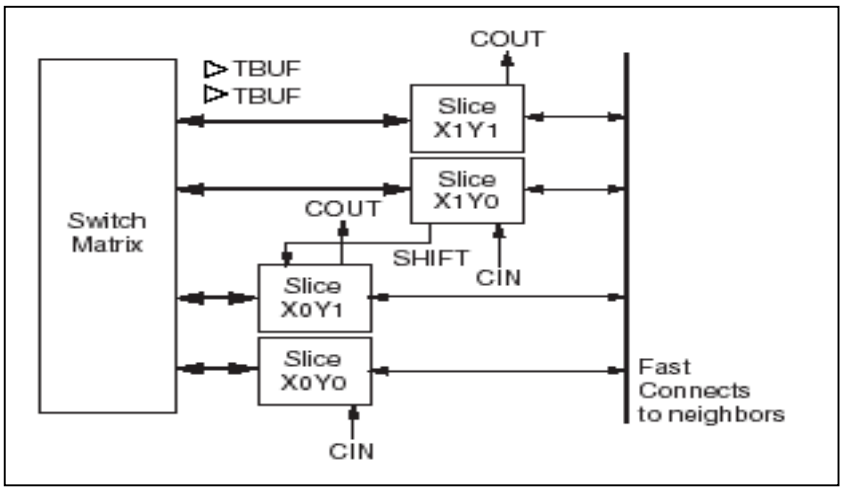

Figure 5.9 : CLB du Virtex-II.

5.2.2.1.2.2. Les IOB_S (Input/Ouput Bloc)

Les blocs d'entrée/sortie (IOB) fournissent une interface entre les broches externes du circuit et la logique interne. Deux IOB_S peuvent être comme une paire différentielle. Celle-ci est souvent connectée à une même matrice d'interconnexions [10], comme cela est illustré sur la figure 5.10.

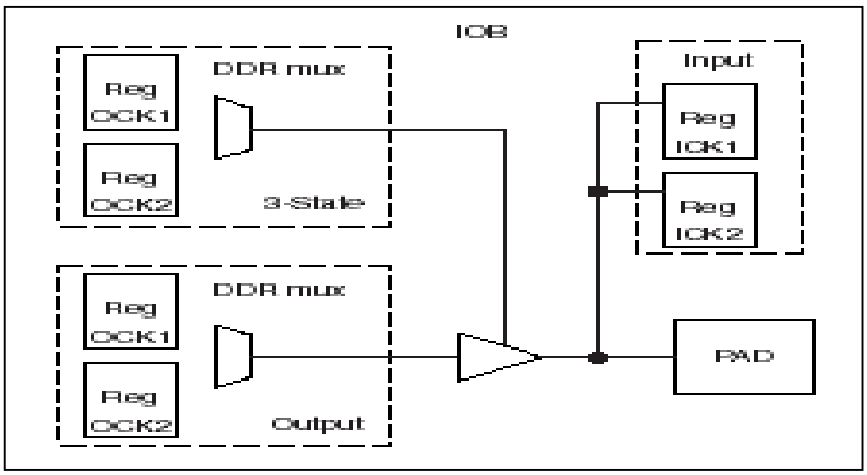

Figure 5.10 : IOB du Virtex-II.

5.2.2.1.2.3. Le Multiplieur 18bits ×18 bits

La famille Virtex-II incorpore des blocs Multiplieurs 18bits×18bits. L'implémentation d'un multiplieur sur cette famille peut être effectuée selon deux manières, la première est l'utilisation directe des blocs multiplieurs, la seconde est la configuration des CLB en multiplieur. Cette tâche est réalisée grâce à un outil nommé Corgen IP qui nous permet de générer des modules pré optimisés. Celle-ci est jugée moins efficace par rapport à la première, vue que l'utilisation des CLB présente une consommation d'énergie [10].

La disposition des blocs multiplieurs se trouve implémentée juste à coté des blocs Select RAM. Cette technique permet d'augmenter les performances d'une application si les opérandes proviennent d'une mémoire.

Ces blocs multiplieurs permettent d'effectuer une multiplication sur des opérandes signés ou non signés et avoir un résultat sur 36 bits, ils peuvent aussi avoir des entrées sorties régulières.

Le schéma du bloc multiplieur 18bits×18bits est montré sur la figure 5.11.

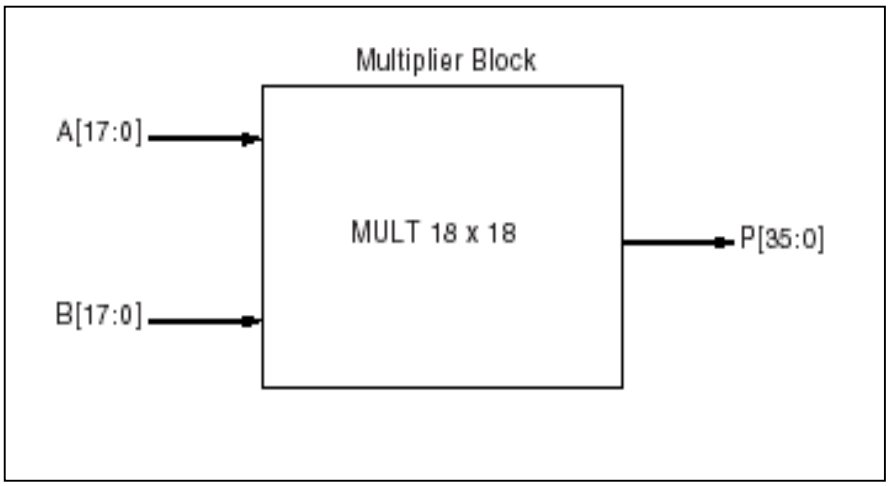

Figure 5.11 : Bloc Multiplieur 18bits ×18bits.

5.2.2.1.2.4. Le DCM (Digital Clock Manager)

Le bloc DCM fournit des solutions auto-étalonnées, pour la compensation du temps de l'horloge, multiplication de l'horloge et division, c'est-à-dire, possibilité de

diminuer le temps de montée d'une horloge et d'éliminer les pertes dues aux routages. Le DCM utilise un routage particulier dédié qui permet une bonne précision pour le contrôle de l'horloge. Le routage en question est composé par huit lignes par quadrant. Le DCM est représenté sur la figure 5.12 [10].

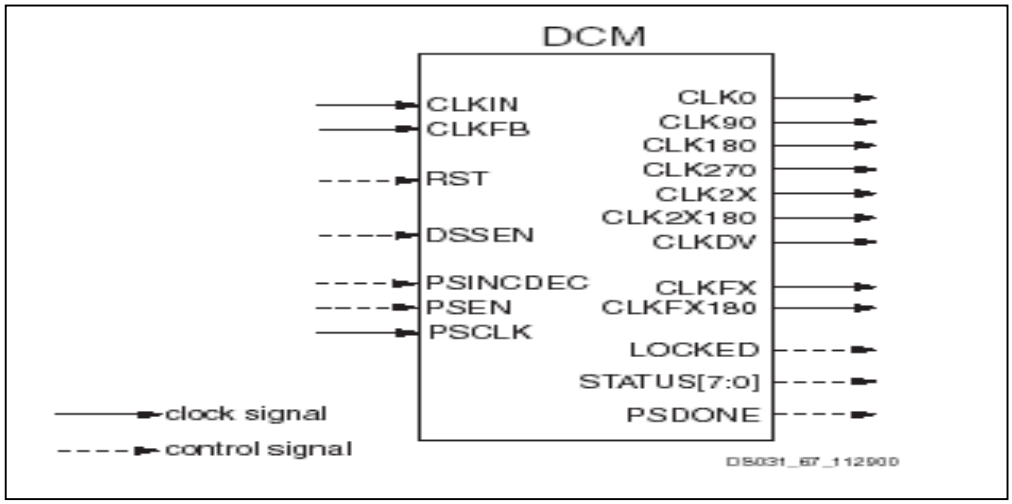

Figure 5.12 : Bloc DCM du Virtex-II.

5.2.2.1.2.5. Bloc Select RAM A 18 K bit

Le Virtex-II inclut dans son architecture un nombre important de blocs mémoires de 18 Kbits. Ceci augmente l'espace mémoire du circuit, si les CLB_S sont configurés comme des mémoires. Chaque bloc Select RAM à double port est de taille de 2×18 Kbits. Ces derniers peuvent avoir indépendamment les signaux d'horloges et de commandes [10]. D'une manière générale les blocs Select RAM (SRAM) sont implémentés suivant les configurations suivantes [11] :

- Lire après écriture (Read After Write).
- Lire avant écriture (Read Before Write).
- Lire seulement (Read Only).

Ces blocs mémoires peuvent être utilisés comme des RAM a simple ou double ports, et ils supportent plusieurs configurations.

Ces configurations sont déterminées par un compromis entre la taille de la donnée et le nombre d'adresse ligne. Les schémas blocs des deux types (simple et double port) sont montrés respectivement sur les figures 5.13 et 5.14.

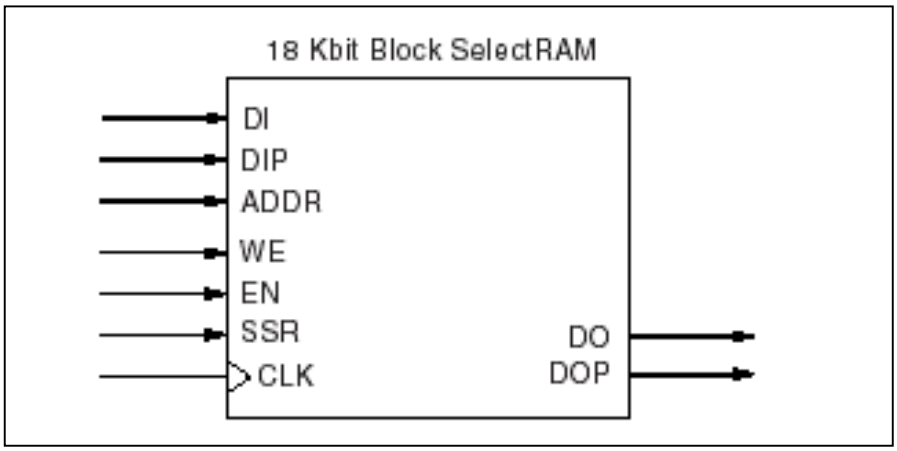

Figure 5.13 : Bloc Select RAM à simple port.

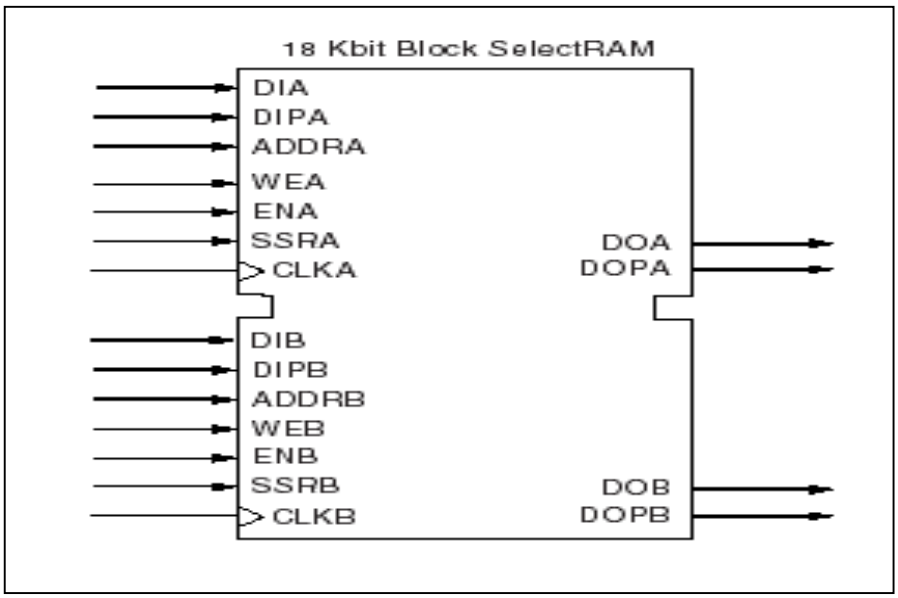

Figure 5.14 : Bloc Select RAM à double port.

5.3. Approche de conception des FPGAs

L'approche de conception des FPGAs est basée sur la synthèse avec le VHDL, la méthodologie de conception est illustrée dans la figure 5.15. Une architecture de conception est alors développée pour doter le comportement hardware en description

VHDL. Comme outil de conception adapté, nous avons utilisée dans notre travail "ISE Foundation" (Integrated Software Engineering), ce dernier réalise automatiquement la synthèse, la simulation et l'implémentation. Une description détailler d'outil ISE sera présentée dans la section suivante:

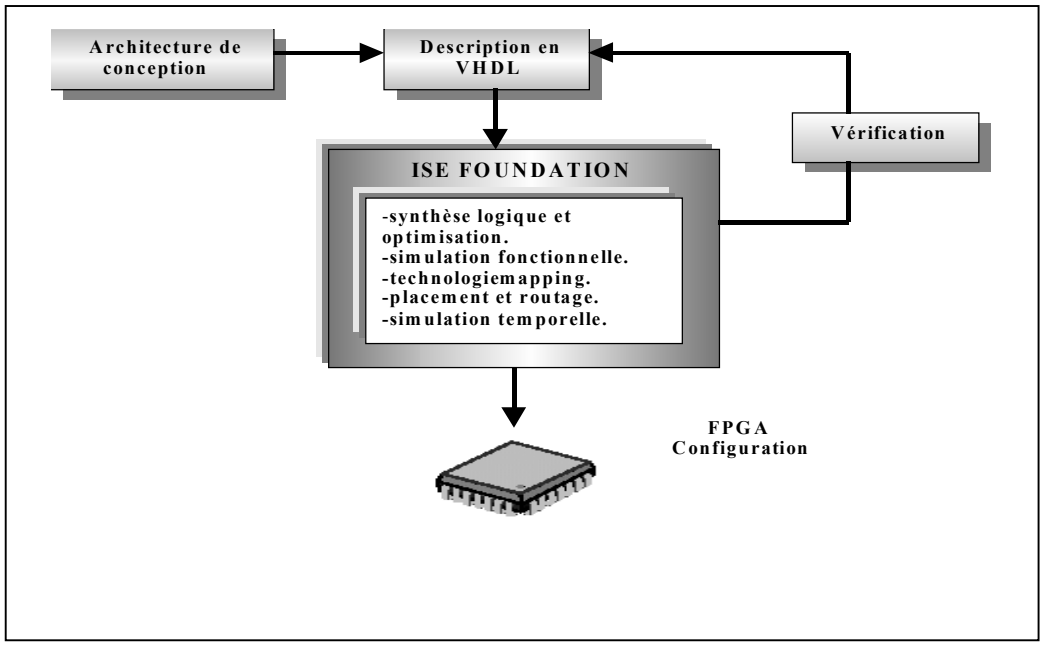

Figure 5.15 : Méthodologie de conception sur circuit FPGA.

La conception se déroule principalement en trois phases:

- La saisie du circuit.
- L'implémentation.
- La configuration du composant.

Aux quelles il faut ajouter les phases de vérifications:

- La simulation fonctionnelle.
- La simulation temporelle.

5.3.1. L'outil de conception ISE Foundation

ISE Foundation est un panel d'outils ISE développé et commercialisé par la société XILINX. Ce panel est un pacquage complet qui intègre toutes les étapes de conception d'un circuit intégré (FPGA) d'une manière automatique. Ce dernier contient plusieurs modules qui fonctionnent selon un processus itératif (Fig 5.16).

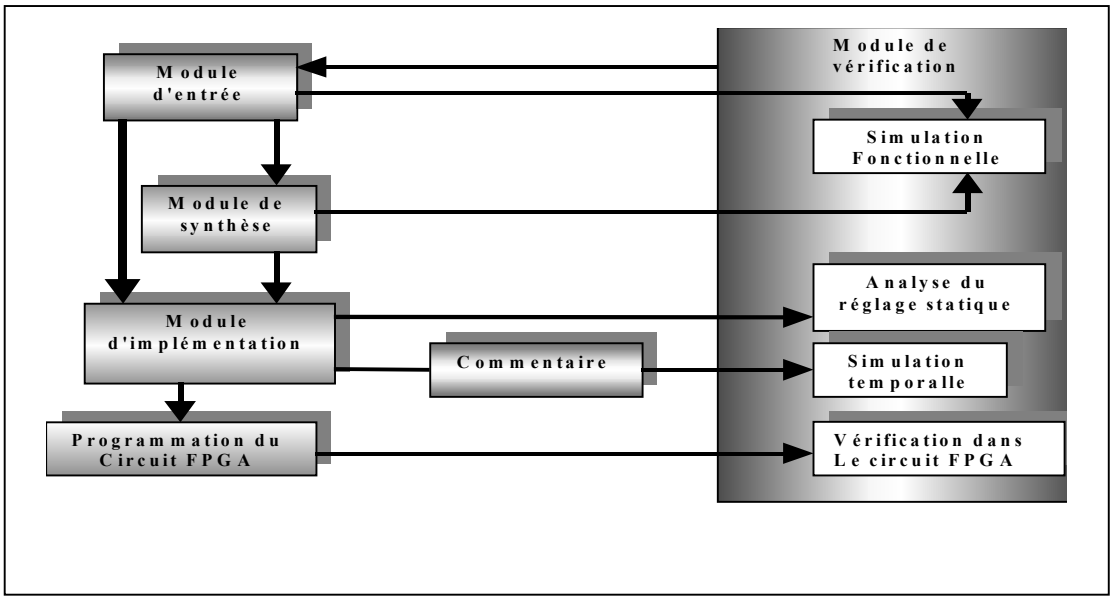

Figure 5.16 : Les modules de l'outil ISE Foundation.

5.3.2. Module d'entrée

Ce module permet la description d'une architecture ou d'un dessin soit en code VHDL, un éditeur de schéma ou bien le diagramme d'état.

5.3.3. Module de synthèse

Il transforme la description VHDL en porte logique (description proche des ressources matérielles) et il fait l'optimisation (le signaux inutilisés sont retirés, les expressions booléennes sont simplifiées, les signaux équivalent sont détectés).

5.3.4. Module d'implémentation

Ce module réalise le « Mapping », c'est-à-dire la projection des équations sur les différents blocs de circuit. Ainsi, il réalise le placement/ routage qui consiste à attribuer les blocs du circuit à chaque équation délivrée par la projection et il définit les connexions.

L'algorithme de placement, place physiquement les différentes cellules et les chemins d'interconnexions dessinés entre les cellules afin de faciliter le routage [13].

5.3.5. Module de vérification

Il réalise la simulation et l'analyse de temps d'exécution (Static Timing Analysis). La simulation se déroule en parallèle après chaque bloc, pour vérifier que le dessin est opérationnel. Elle se fait en deux étapes:

- Après la description en code VHDL on procède à la simulation fonctionnelle.
- Après l'implémentation on procède à la simulation temporelle.

5.3.6. La programmation du circuit FPGA

Ce module réalise la programmation du circuit FPGA, il est lié au module vérification pour une vérification final du circuit.

5.4. Conclusion

Dans ce chapitre nous avons spécialisé notre étude sur le type FPGA en donnant une description détaillée de la famille Virtex-II de Xilinx, pour laquelle notre choix s'est porté pour l'implantation de notre architecture, à cause des multiples avantages offerts. En dernière étape nous avons présenté l'approche de conception des FPGAs, comme outil de conception adapté-le "ISE Foundation" (Integrated Software Engineering), ce dernier permet de réaliser automatiquement la synthèse, la simulation et l'implémentation de notre architecture.

Dans le chapitre suivant, nous allons présenter les résultats de la simulation et l'implémentation de notre architecture.

CHAPITRE 6

SIMULATION ET IMPLEMENTATION DE L'ARCHITECTURE

6.1. Introduction

Après avoir élaboré l'architecture du multiplieur à précision variable, nous allons procéder à une autre étape essentielle dans le processus de conception en électronique, qui est la description, la simulation et enfin l'implémentation sur circuit FPGA de la famille Virtex-II de Xilinx. Toutes ces étapes vont être réalisées dans l'environnement de conception "fondation 4.1" de Xilinx.

Ces dernières peuvent être résumées comme suit :

- Description de l'architecture globale et des cellules élémentaires en langage VHDL par l'outil PROJECT Navigator qui inclut aussi un système nommé CORE Generator. Celui ci nous à permis de générer les composants élémentaires sous forme de macro tels que les compteurs, le multiplieur, les additionneurs, les comparateur, etc.
- Simulation de l'architecture par l'outil MODEL Sim.
- Implémentation de l'architecture sur circuit Virtex-II par l'outil FPGA express.

6.2. Description VHDL

Avant d'entamer la simulation, nous allons décrire notre architecture à l'aide du langage VHDL, l'utilisation de ce langage nous a permis la création d'une bibliothèque de travail qui comporte tous les éléments nécessaires pour la conception de notre architecture. Nous allons décrire en premier lieu les cellules élémentaires qui vont être utilisées pour concevoir des macros d'un niveau d'abstraction plus haut, par conséquent notre bibliothèque comporte deux types de cellules :

* Cellules élémentaires.

*Cellules complexes.

Les cellules élémentaires décrites en langage VHDL et les macros générés par le système Core Generator sont données respectivement dans les tableaux 6.1 et 6.2.

Cellule	Fonction
AND2_1	Porte AND à 2 entrées d'un bit.
ANDI2_1	Porte AND à 2 entrées d'un bit dont une est inversée.
OR2_1	Porte OR à deux entrées d'un bit.
OR1_6	Porte OR à une entrée de 6 bits.
INV1	Inverseur d'un bit.
XOR2_1	Porte XOR à 2 entrées d'un bit.
LATCH1	Bascule D à 1 bit.
RELATCH2	Registre à 2 bits et à commande «reset» et «enable»
TRS1	Porte TRS (on, off, haut impédance) à une entrée d'un bit.
TRS64	Porte TRS (on, off, haut impédance) à une entrée de 64 bits.
INOUT1	Interconnexion entrée 1 bit/sortie 1 bit.
INOUT66s2_64	Interconnexion entrée 66 bits/sorties 2 et 64 bits
INOUT1_64s65	Interconnexion entrées 1 et 64 bits/sortie 65 bits
INOUT2_62s64	Interconnexion entrées 2 et 62 bits/sortie 64bits
CLKDLL	Circuit DLL.

Tableau 6.1 : Les cellules élémentaires de l'architecture décrites en langage VHDL.

Cellule	Fonction
MUL64	Multiplieur de 64×64 bits.
RMUX2_2	Multiplexeur à 2 entrées de 2 bits et à sortie registrée.
MUX2_7	Multiplexeur à 2 entrées de 7 bits.
MUX2_64	Multiplexeur à 2 entrées de 64 bits.

Cellule	Fonction
RMUX2_64	Multiplexeur à 2 entrées de 64 bits et à sortie registrée.
RAMA64_64	Mémoire RAM (pour le multiplicande A) de 64 mots de 64 bits.
RAMB64_64	Mémoire RAM (pour le multiplieur B) de 64 mots de 64 bits.
RAM128_64	Mémoire RAM de 128 mots de 64 bits
RCMP2_6	Comparateur à 2 entrées de 6 bits et à sortie registrée
RCMP2_7	Comparateur à 2 entrées de 7 bits et à sortie registrée
CMPT6	Compteur binaire à 6 bits.
CMPT7	Compteur binaire à 7 bits.
ADD6	Additionneur à 6 bits à sortie non registrée.
RADD6	Additionneur à 6 bits et à sortie registrée.
RADD64	Additionneur à 64 bits et à sortie registrée.
RSADD64	Additionneur à 64 bits et à sorties registrées et séparées.
RADD65	Additionneur à 65 bits et à sortie registrée.

-b-

Tableau 6.2 (a-b) : Les cellules élémentaires de l'architecture générées par le système CORE Generator.

Les cellules complexes sont données dans le tableau 6.3.

Cellule	Fonction
MADG	Module d'adressage
MMUL	Module de multiplication
MACC	module d'accumulation
MGSH	Module générateur des signaux d'horloge
MGSC	Module générateur des signaux de commande

Tableau 6.3 : Les cellules complexes (ou macros) de l'architecture.

La description VHDL des modules (d'adressage, de multiplication, d'accumulation, générateur des signaux d'horloge et générateur des signaux de commande) ainsi que la partie opérative, la partie contrôle et l'architecture globale sont données en annexe A.

6.3. Simulation

Cette étape permet la vérification de la fonctionnalité de l'architecture établie, elle permet la détection des erreurs si elles existent et d'apporter des corrections. Pour réaliser cette étape nous allons utiliser l'outil de simulation MODEL Sim.

La simulation de notre architecture se fait de façon hiérarchique, cela veut dire qu'on effectue la simulation des cellules élémentaires (comme les multiplexeurs, porte AND, porte XOR,... etc.) avant de faire la simulation des macro et enfin l'architecture globale. Autrement dit, la simulation se fait du niveau d'abstraction le plus bas vers le niveau le plus haut. Cette façon hiérarchique est la meilleure manière pour simuler n'importe qu'elle architecture.

La procédure de simulation de l'architecture du multiplieur à précision variable est illustrée par l'arbre de simulation donnée sur la figure 6.1.

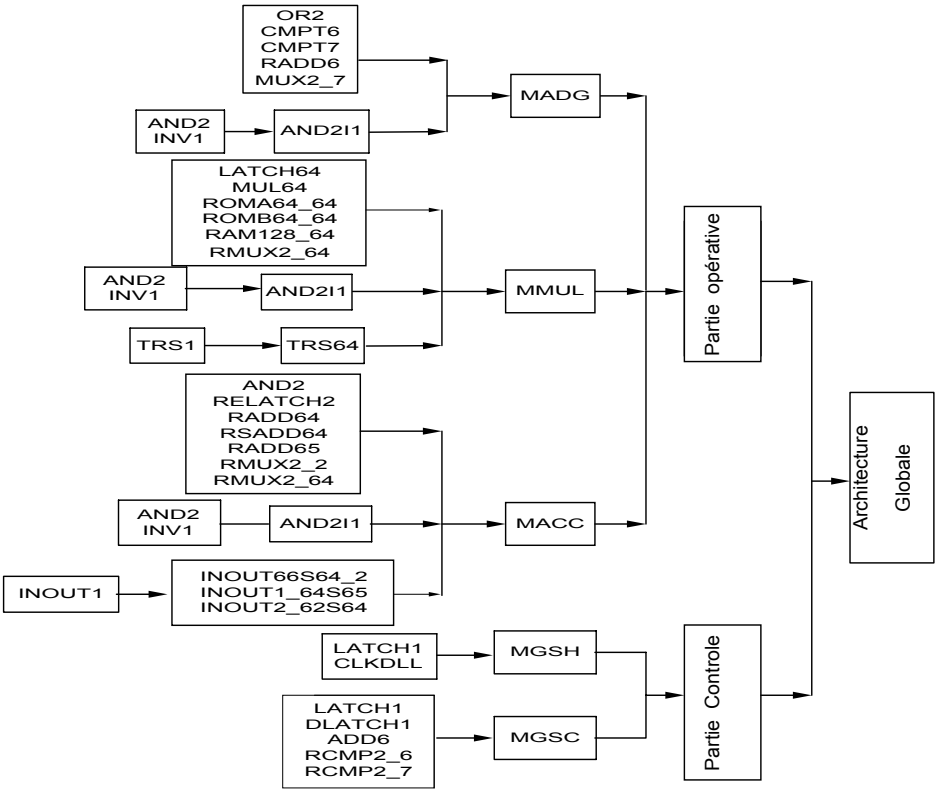

Figure 6.1 : Arbre de simulation de l'architecture du multiplieur à précision variable.

6.3.1. Simulation des différents modules de l'architecture

Après avoir décrit en VHDL l'architecture de chaque module constituant le multiplieur à précision variable, on définit en premier lieu les vecteurs de test sur l'outil HDL Bencher. Ces vecteurs contiennent des valeurs qui seront utilisées comme données d'entrées pour le module à simuler. Une fois cette étape achevée, l'architecture subit une simulation sur l'outil Model Sim. Les résultats de la simulation obtenus sont illustrés par les chronogrammes des figures 6.2, 6.3, 6.4 et 6.5, ils sont données par l'éditeur WAVE forme.

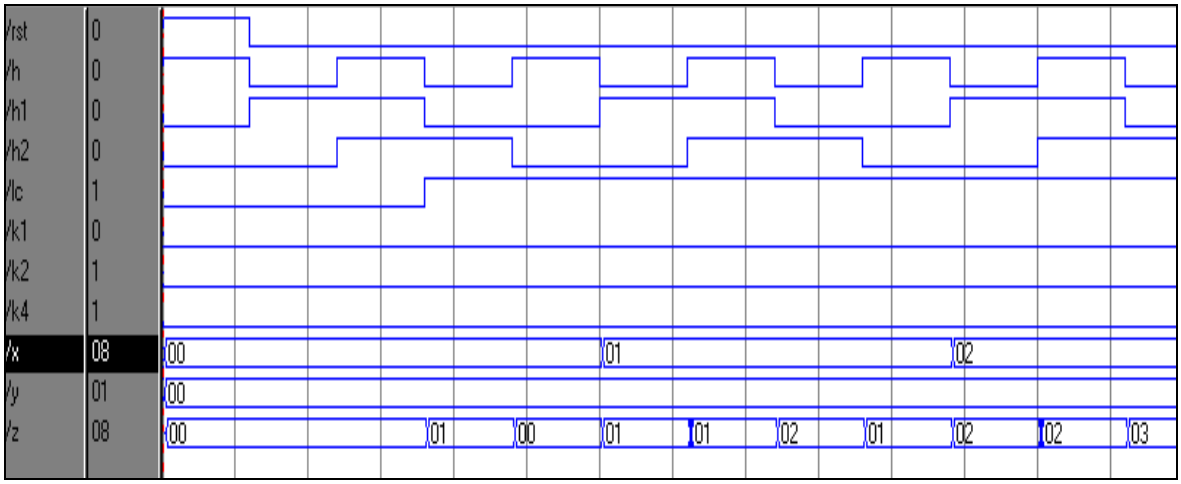

Figure 6.2 : Chronogrammes de simulation du module d'adressage.

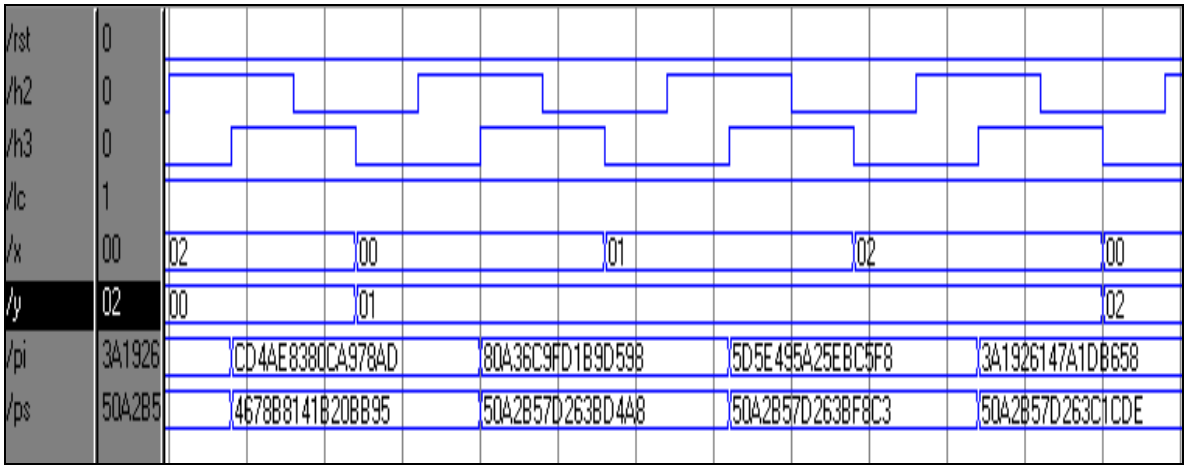

Figure 6.3 : Chronogrammes de simulation du module de multiplication.

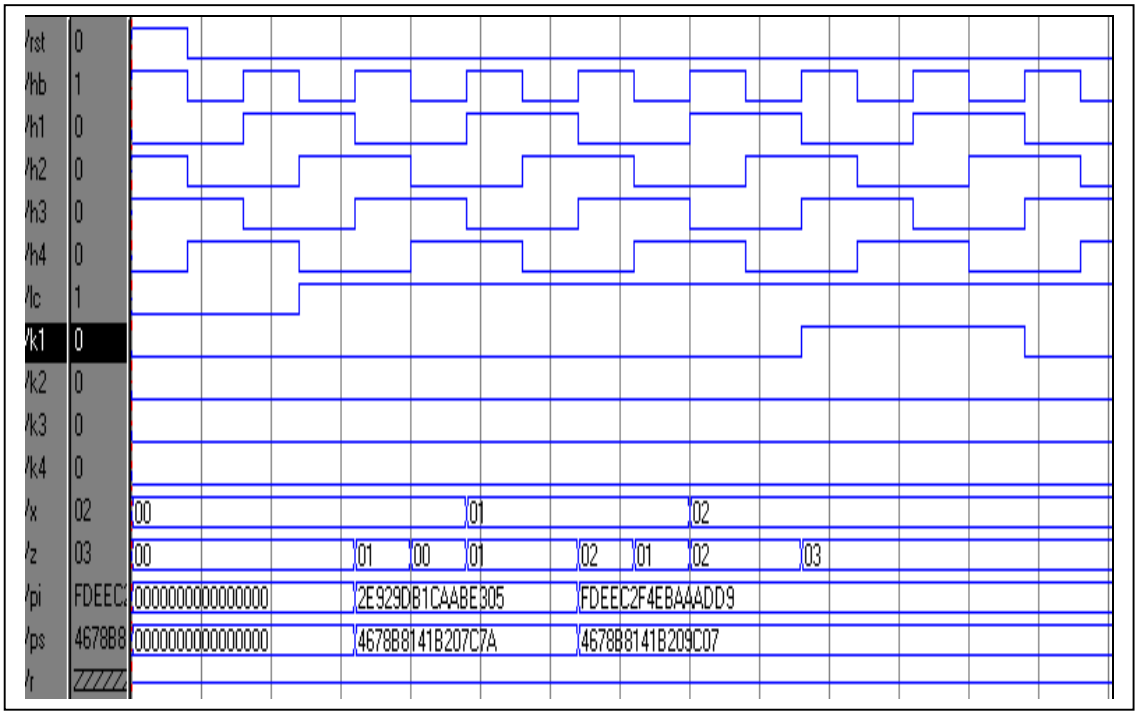

Figure 6.4 : Chronogrammes de simulation du module d'accumulation.

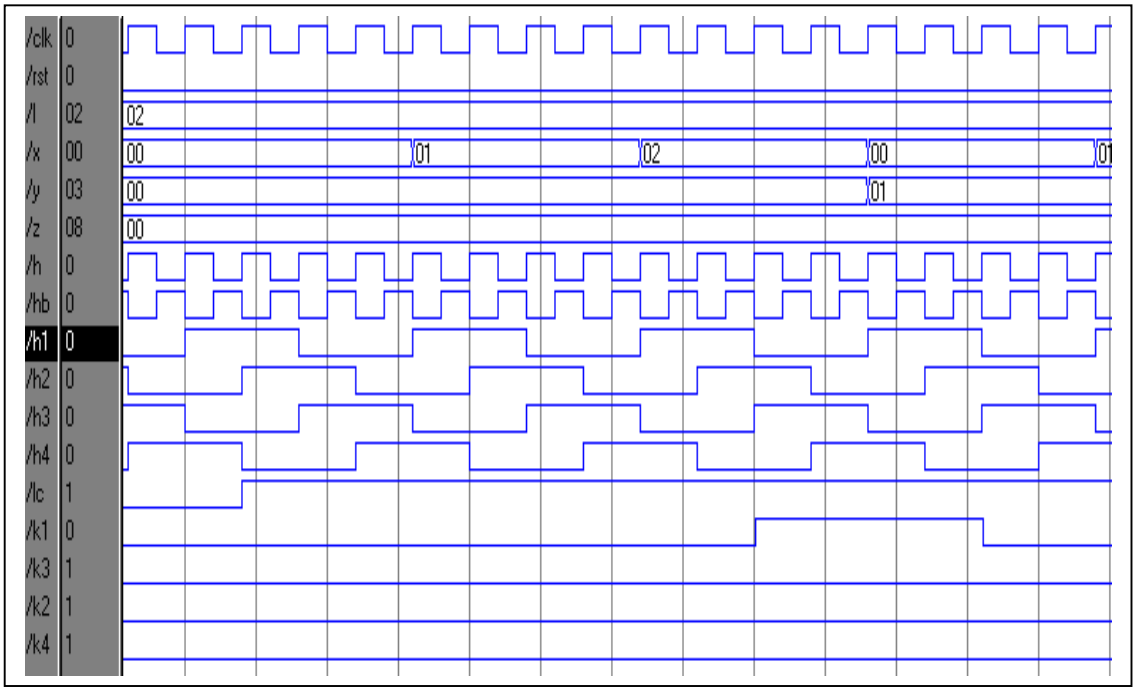

Figure 6.5 : Chronogrammes de simulation de la partie contrôle.

6.3.2. Simulation de l'architecture globale

Après avoir vérifié la fonctionnalité de tous les modules de l'architecture, il reste à vérifier le bon fonctionnement de l'architecture globale. Pour ce faire, on définit en premier lieu les vecteurs de test (les valeurs choisis des signaux et données à l'entrée de ce bloc) sur l'outil HDL Bencher. Une fois cette étape achevée, l'architecture subit une simulation sur l'outil Model Sim. Les résultats de la simulation obtenus sont illustrés par les chronogrammes de la figure 6.6, ils sont données par l'éditeur WAVE forme.

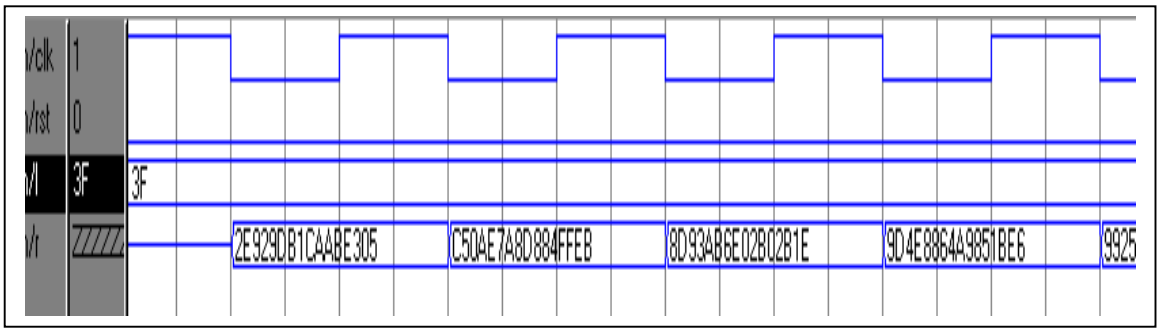

Figure 6.6 : Chronogrammes de simulation de l'architecture globale.

Le résultat de la simulation de l'architecture globale, ainsi que les résultats de la simulation des principaux étages constituant cette dernière, obtenus sur les figures de cette première étape son acceptables et compatibles aux résultats théoriques prévus, ceci confirme le bon fonctionnement de cette architecture et permet le passage à la dernière étape à savoir l'implémentation.

6.4. Implémentation

Dans cette étape on a réalisé l'implémentation physique de l'architecture conçue sur le circuit FPGA ciblé qui est le XC 2V 1000-5bg575 du Virtex-II, les caractéristiques de ce circuit sont :

- Le nombre de CLB est de 32×40=1280 CLB (1CLB contient 4 Slices)
- La vitesse d'exécution est de -5

L'outil d'implémentation utilisé est FPGA Express. Ce dernier permet aussi la création des fichiers résultats qui nous donnent des informations sur les ressources utilisées (nombre de CLB, de blocs mémoires, DCM_S...) ainsi que le temps d'exécution de l'architecture. L'espace qu'occupe l'architecture sur le circuit FPGA XC 2V 1000 ainsi que le routage d'interconnexion sont montrés sur les figures 6.7 et 6.8.

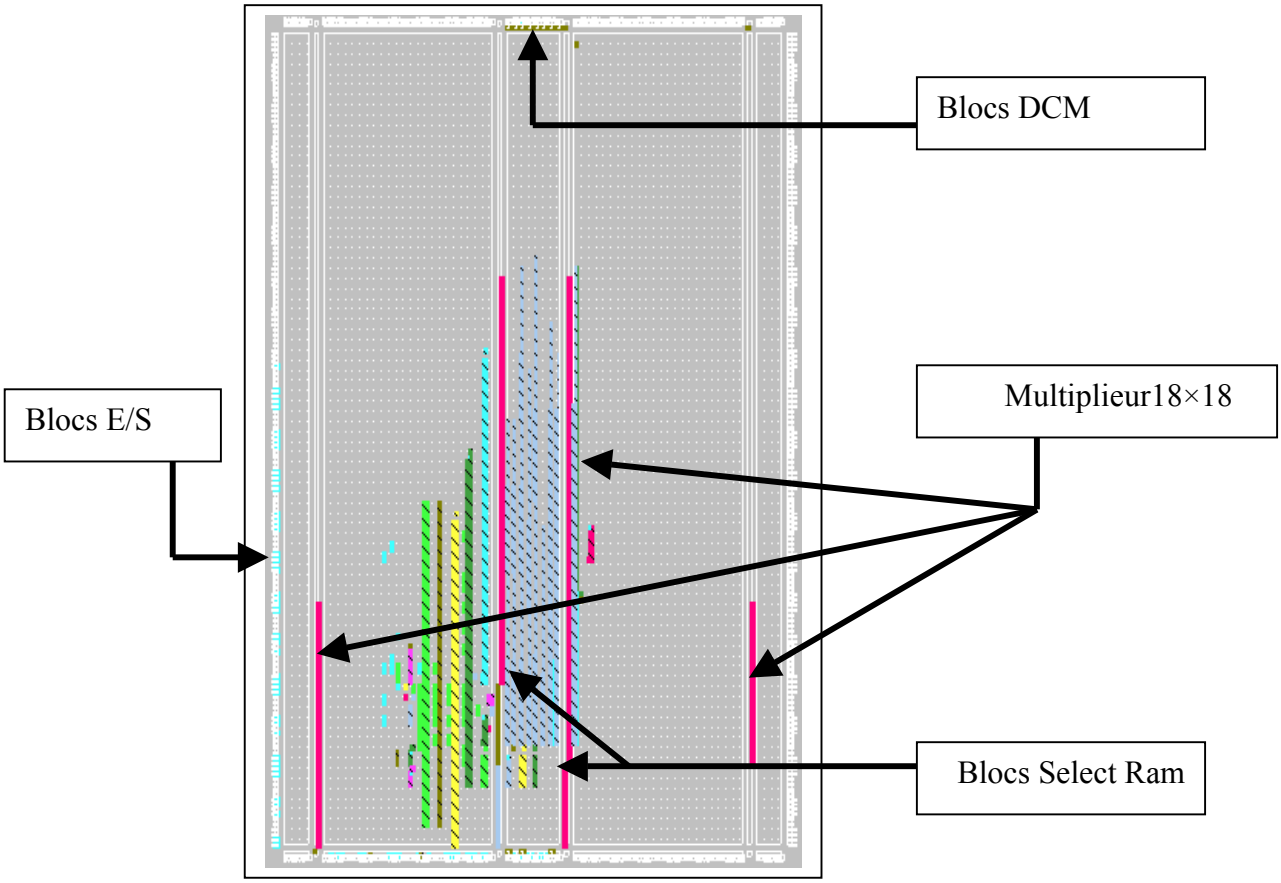

Figure 6.7 : L'espace qu'occupe l'architecture sur le circuit FPGA XC 2V 1000-5bg575.

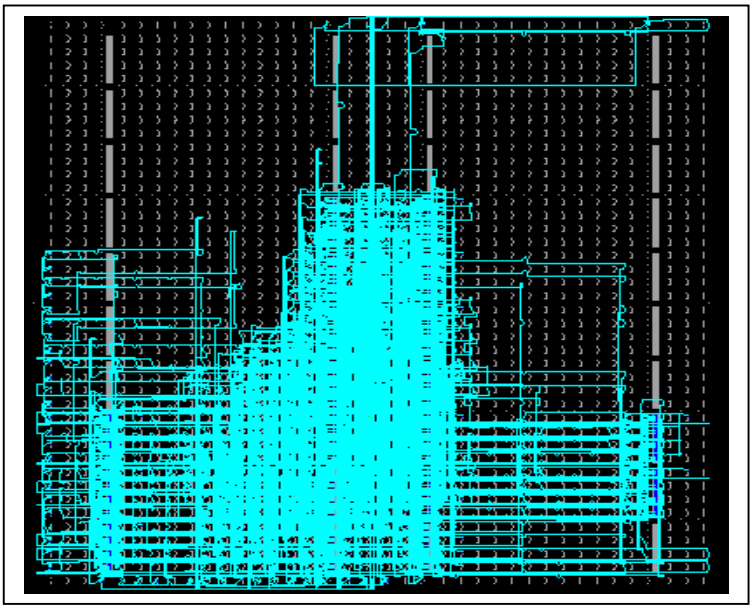

Figure 6.8 : Le routage et le maping du circuit FPGA XC 2V 1000-5bg575.

6.4.1.Taux d'occupations

Les taux d'occupations correspondants à l'implémentation de l'architecture sur le circuit XC 2V1000-5bg575 du Virtex II sont illustrées sur le tableau 6.4, ces résultats sont donnés par l'éditeur "place route report".

Elément	Tau d'occupation	%
IOB	72 / 328	21%
RAMB16	3 / 40	7%
SLICE	596 / 5120	11%
MUL18×18	16 / 40	40%
BUFGMUX	11 / 16	68 %
DCM	3 / 8	37 %

Tableau 6.4 : Les taux d'occupations lors de l'implémentation de l'architecture sur le circuit XC2v1000du Virtex-II .

6.4.2. Temps d'exécution

Le temps d'exécution de l'architecture (T_e) est calculé comme étant le produit du nombre de cycles ($4n^2$) et le temps d'un cycle (T_c), ce dernier correspond au délai du chemin critique (le temps de propagation à travers le plus lent des blocs de la logique combinatoire constituant la partie traitement de l'architecture (voir figure 4.31)), il est déterminé par l'outil TIMING Analyser. Un schéma significatif du chemin critique est montré sur la figure 6.9. Ce chemin est composé de :

- La mémoire (MA)
- Le multiplieur (MUL)
- Le routage d'interconnexion (r)

D'où le temps de calcul de ce chemin est :

$T_c = T_{MA}$(temps d'accès à la mémoire) + T_R (temps du routage d'interconnexion) + T_{MUL} (délais du multiplieur) = 33 ns.

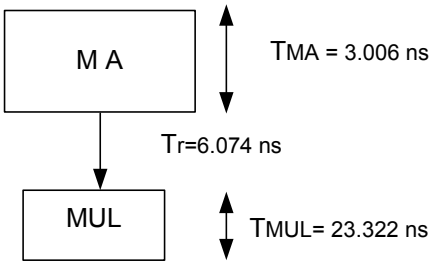

Figure 6.9 : Chemin critique de l'architecture.

6.5. Conclusion

Dans ce chapitre, nous avons présenté en premier lieu les résultats de la simulation des différents modules de notre architecture, puis nous avons présenté le résultat de la simulation de l'architecture globale. Les résultats obtenus lors de cette étape ont permis de confirmer la bonne fonctionnalité de cette architecture.

Dans la deuxième étape, nous avons donné les résultats (taux d'occupations et temps d'exécution) obtenus lors de l'implémentation de l'architecture sur le circuit FPGA ciblé qui est le XC2v1000-5bg575 du Virtex-II. Les résultats finaux obtenus lors de cette étape ont permis de montré la performance de cette architecture.

Les étapes réalisées dans ce chapitre, à savoir la simulation et l'implémentation nous ont été d'un apport très bénéfique.

CONCLUSION

Dans le cadre de notre travail, nous avons conçu l'architecture du multiplieur en virgule flottante à précision variable.

Nous avons pris connaissance tout d'abord des arithmétiques mises au point pour améliorer la qualité des résultats d'un calcul numérique. L'étude de chacune d'elles nous a permis non seulement d'élargir nos connaissances dans le domaine mais aussi de constater les principales différences entre ces méthodes.

L'arithmétique virgule flottante à précision variable étant adoptée nous nous sommes consacrés dans une deuxième phase à l'élaboration de l'algorithme de multiplication de nombres en virgule flottante à précision variable, ensuite à la conception et la mise en œuvre des différents modules de l'architecture permettant d'exécuter cet algorithme et qui sont rapportés dans le chapitre 4.

Après avoir élaboré l'architecture du multiplieur à précision variable, nous avons abordé les étapes de description de l'architecture en langage VHDL, la simulation et l'implémentation de l'architecture, toutes ces étapes ont été réalisées dans l'environnement de conception "fondation 4.1" de Xilinx. Les différents modules constituant cette architecture ont été générés d'une manière optimisée grâce à un système nommé CORE generator et qui est dédié aux circuits $FPGA_S$ de Xilinx. Afin de garantir un fonctionnement correct de l'architecture, cette dernière a été synthétisée et ensuite simulée par l'outil MODELSIM, le résultat obtenu au cour de cette étape est satisfaisant et concorde avec celui prévu et calculé en utilisant l'outil MAPLE 6 (voir annexe B).

Pour l'implémentation de notre architecture, nous avons utilisé un circuit FPGA de Xilinx appartenant à la famille Virtex-II, nous avons exploité les ressources internes disponibles sur ce type de circuit, tel que les SLICES, les RAMS, les DCMS, etc. Comme résultat, il a été montré que l'implémentation de l'architecture a donné un temps de cycle de 33 ns.

Incontestablement, ce travail nous a ouvert la voie vers un domaine en évolution permanente qui est celui de l'arithmétique des ordinateurs et l'architecture des systèmes ainsi que l'utilisation d'un langage de description matériel de haut niveau en l'occurrence le "VHDL" et nous a permis aussi d'acquérir des connaissances de la technologie FPGA en générale et du circuit Virtex-II en particulier.

Nous espérons que ce travail a apporté une modeste contribution dans le domaine de l'arithmétique et de l'architecture des systèmes et un perfectionnement ultérieur (plus éloigné) pouvant être ajouté qui permettra de manipuler d'autres opérations à précision variable, parmis lesquelles :

1. L'addition et la soustraction en précision variable, les champs mantisses doivent être alignés d'abord selon les champs d'exposants. Des circuits de décalage additionnel sont exigés à l'alignement des champs mantisses et de renormalisation.
2. La division et le calcul de la racine carrée en précision variable.
3. La capacité de supporter les différents modes d'arrondis.
4. La capacité de manipuler les exceptions (infini, nombres "dénormalisés", etc.)

APPENDICE A
DESCRIPTIONS VHDL

-----La description en langage VHDL du module d'adressage se fait de la manière suivante :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unisigned.all;
entity MADG is
  Port ( RST, H, H1, H2, LC, K1, K2, K4)  : in std_logic;
         X : out std_logic_vector(5 downto 0);
         Y : out std_logic_vector(5 downto 0);
         Z : out std_logic_vector(6 downto 0));
end MADG;
architecture structural of MADG is
component OR2 is
  Port ( X : in std_logic;
         Y : in std_logic;
         Z : out std_logic);
end component;
component AND2I1 is
   Port ( X : in std_logic ;
          Y : in std_logic ;
          Z : out std_logic) ;
     end component;
component XOR2 is
  Port ( X : in std_logic ;
         Y : in std_logic ;
         Z : out std_logic) ;
end component;
```

```
component CMTR6
      port (Q : out std_logic_vector (5 downto 0) ;
        CLK : in std_logic ;
          CE : in std_logic ;
       SCLR : in std_logic) ;
end component;
component inout1_6s7 is
  port (xin : in std_logic;
          yin : in std_logic_vector(5 downto 0);
         zout : out std_logic_vector(6 downto 0));
end component;
component RADD6 is
   port (A : in std_logic_vector (5 downto 0) ;
           B : in std_logic_vector (5 downto 0) ;
     C_IN : in std_logic ;
          Q : out std_logic_vector (6 downto 0) ;
      CLK : in std_logic ;
   ACLR : in std_logic) ;
end component;
component CMTR7 is
port (Q : out std_logic_vector(6 downto 0);
    CLK : in std_logic;
     CE : in std_logic;
 SCLR : in std_logic);
end component;
component MUX2_7 is
   port (MA : in std_logic_vector (6 downto 0) ;
           MB : in std_logic_vector (6 downto 0) ;
              S : in std_logic_vector (0 downto 0) ;
             O : out std_logic_vector (6 downto 0)) ;
end component;
signal N1 : std_logic;
signal N2 : std_logic;
signal N3 : std_logic_vector (5 downto 0);
```

```
signal N4 : std_logic_vector (5 downto 0);
signal N5 : std_logic_vector (5 downto 0);
signal N6 : std_logic_vector (5 downto 0);
signal N7 : std_logic;
signal N8 : std_logic_vector (5 downto 0);
signal N9: std_logic;
signal N10 : std_logic_vector (6 downto 0);
signal N11 : std_logic_vector(6 downto 0) ;
signal GND : std_logic ;
signal VCC : std_logic ;
begin
GND<='0' ;
VCC<='1' ;
U1 : OR1 port map (X=>RST, Y=>K1, Z=>N1) ;
U2 : AND2I1 port map (X=> K2, Y=>L2, Z=>N2);
U3 : CMTR6 port map (SCLR=>N1, CLK=>H1, CE=>N2, Q=>N3);
U4 : CMTR6 port map (SCLR=>RST, CLK=>H1, CE=> K1, Q=>N6);
U5 : RADD6 port  map (CLK=>H2, ACLR=>RST, A=>N3, B=>N4, C_IN=>GND,
Q=>N5) ;
U6 : RADD6 port map (CLK=>H2, ACLR=>RST, A=>N3, B=>N4, C_IN=>VCC,
Q=>N6);
U7 : XOR1 port map(X=>H1,Y=>H2, Z=>N7);
U8: MUX2_7 port map (MA=>N5, MB=>N6, S(0)=> N7, O=>N8);
U9 : AND2I1 port map (X=> K4, Y=> K2, Z=>N9);
U10 : CMTR7 port map (SCLR=>RST, CLK=>H, CE=>N9,Q=>N10);
U11: MUX2_7 port map (MA=>N8, MB=>N10, S(0)=> K2, O=>Z);
X<=N3 ;
Y<=N4 ;
end structural;
```

------La description en langage VHDL du module de multiplication se fait de la manière suivante :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unisigned.all;
entity MMUL is
  Port ( RST, LC, H2, H3) : in std_logic ;
         X : in std_logic_vector (5 downto 0) ;
         Y : in std_logic_vector (5 downto 0) ;
         PI : out std_logic_vector (63 downto 0) ;
         PS : out std_logic_vector (63 downto 0) ;
 end MMUL;
architecture structural of MMUL is
component MUL64
   port (A : in std_logic_vector(63 downto 0);
         B : in std_logic_vector (63 downto 0) ;
         Q : out std_logic_vector (127 downto 0)) ;
end component;
component inout128s64_64 is
   Port ( XIN : in std_logic_vector (127 downto 0) ;
          YOUT: out std_logic_vector (63 downto 0) ;
         ZOUT : out std_logic_vector (63 downto 0) ) ;
end component;
component RLatch64 is
   Port ( CLK : in std_logic;
            CLR : in std_logic;
             D : in std_logic_vector (63 downto 0) ;
            Q : out std_logic_vector (63 downto 0) ) ;
end component;
component tablea3w_64 is
   port (ADDR : in std_logic_vector (5 downto 0) ;
             CLK : in std_logic;
          DOUT : out std_logic_vector (63 downto 0);
```

```
            EN : in std_logic) ;
end component;
component tableb3w_64 is
   port (ADDR : in std_logic_vector (5 downto 0) ;
            CLK : in std_logic;
           DOUT : out std_logic_vector (63 downto 0);
             EN : in std_logic) ;
end component;
signal N1 : std_logic_vector (63 downto 0) ;
signal N2 : std_logic_vector(63 downto 0) ;
signal N3 : std_logic_vector(127 downto 0);
signal N4 : std_logic_vector(63 downto 0);
signal N5 : std_logic_vector(63 downto 0);
begin
U1: tablea64w_64b port map (CLK=>H2, EN=>Lc, ADDR=>X, DOUT=>N1) ;
U2: tableb64w_64b port map (CLK=>H2, EN=>Lc, ADDR=>Y, DOUT=>N2) ;
U3: MUL64 port map (A=>N1, B=>N2, O=>N3) ;
U4: inout128s64_64 port map (XIN=>N3, YOUT=>N4, ZOUT=>N5) ;
U5: RLatch64 port map (clk=>H3, Reset=>RST, D=>N4, Q=>PI) ;
U6: RLatch64 port map(clk=>H3, CLR=>RST, D=>N5, Q=>PS) ;
end Structural;
```

----La description en langage VHDL du module d'accumulation se fait de la manière suivante :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unisigned.all;
entity MACC is
 Port ( RST,Hb, H1, H2, H3, H4, K1, K2, K3, K4 : in std_logic;
         PI : in std_logic_vector(63 downto 0);
        PS : in std_logic_vector(63 downto 0);
          X : in std_logic_vector(5 downto 0);
          Z : in std_logic_vector(6 downto 0);
```

```
        PI : out std_logic_vector(63 downto 0);
        PS : out std_logic_vector(63 downto 0);
         R : out std_logic_vector(63 downto 0));
end MACC;

architecture structural of MACC is
component inout1_64s65 is
   Port ( xin : in std_logic_vector(63 downto 0);
          yin : in std_logic;
         zout : out std_logic_vector(64 downto 0));
end component;
component inout66s2_64 is
    Port ( xin : in std_logic_vector(65 downto 0);
           yout : out std_logic_vector(63 downto 0);
           zout : out std_logic_vector(1 downto 0));
end component;
component radd64
   port ( A : in std_logic_vector(63 downto 0);
          B : in std_logic_vector(63 downto 0);
       C_IN : in std_logic;
                    CLK : in std_logic;
       SCLR : in std_logic;
          Q : out std_logic_vector(64 downto 0));
end component;
component radd65
   port ( A : in std_logic_vector(64 downto 0);
          B : in std_logic_vector(64 downto 0);
       C_IN : in std_logic;
        CLK : in std_logic;
       SCLR : in std_logic;
            Q : out std_logic_vector(65 downto 0));
end component;
component rsadd64
   port ( A : in std_logic_vector(63 downto 0);
```

```
            B : in std_logic_vector(63 downto 0);
        C_IN : in std_logic;
         CLK : in std_logic;
       ACLR : in std_logic;
Q_C_OUT : out std_logic;
          Q : out std_logic_vector(63 downto 0));
end component;
component rmux2_2
   port (MA : in std_logic_vector(1 downto 0);
          MB : in std_logic_vector(1 downto 0);
           S : in std_logic_vector(0 downto 0);
         CLK : in std_logic;
           Q : out std_logic_vector(1 downto 0));
end component;
component AND1 is
   Port ( X : in std_logic;
          Y : in std_logic;
          Z : out std_logic);
end component;
component OR6 is
   Port ( E : in std_logic_vector(5 downto 0);
          S : out std_logic);
end component;
component RELatch2 is
   Port (CLK : in std_logic;
           EN : in std_logic;
        Reset : in std_logic;
            D : in std_logic_vector(1 downto 0);
            Q : out std_logic_vector(1 downto 0));
end component;
component RLatch64 is
   Port ( clk : in std_logic;
        Reset : in std_logic;
            D : in std_logic_vector(63 downto 0);
```

```
            Q : out std_logic_vector(63 downto 0));
end component;
component mux2_64
   port (  MA : in std_logic_vector(63 downto 0);
           MB : in std_logic_vector(63 downto 0);
           S : in std_logic_vector(0 downto 0);
          O : out std_logic_vector(63 downto 0));
end component;
component RAM128_64 is
   Port ( ADDR : in std_logic_vector(63 downto 0);
            CLK : in std_logic;
            DIN : in std_logic_vector(63 downto 0);
             EN : in std_logic;
             WE : in std_logic;
           DOUT : out std_logic_vector(63 downto 0));
end component;
component TRS64 is
   Port ( X : in std_logic_vector(63 downto 0);
           T : in std_logic;
          Y : out std_logic_vector(63 downto 0));
end component;
component inout2_62s64 is
   Port ( XIN : in std_logic_vector (1 downto 0) ;
           YIN : in std_logic_vector (61 downto 0) ;
         ZOUT : out std_logic_vector (63 downto 0) ) ;
end component;
component inout66s2_64 is
   Port ( E : in std_logic_vector (65 downto 0) ;
          S1 : out std_logic_vector (63 downto 0) ;
          S2 : out std_logic_vector (1 downto 0) ) ;
end component;
component GADD64 is
   port (A : in std_logic_vector (63 downto 0) ;
          B : in std_logic_ vector (63 downto 0) ;
```

```
      C_IN : in std_logic;
           Q : out std_logic_ vector (64 downto 0) ;
       CLK : in std_logic;
     SCLR : in std_logic) ;
end component;
component GADD65 is
    port (A : in std_logic_vector (64 downto 0) ;
            B : in std_logic_ vector (64 downto 0) ;
      C_IN : in std_logic ;
           Q : out std_logic_ vector (65 downto 0) ;
       CLK : in std_logic);
 end component;
 component GADD4 is
    port (A : in std_logic_vector (63 downto 0) ;
            B : in std_logic_ vector (63 downto 0) ;
      C_IN : IN std_logic;
    Q_C_OUT : out std_logic ;
          Q : out std_logic_ vector (63 downto 0) ;
      CLK : in std_logic;
   ACLR : in std_logic) ;
end component;
component OR6 is
    Port ( E : in std_logic_vector (5 downto 0);
            S : out std_logic);
end component;
component ERLatch2 is
    Port (CLK : in STD_LOGIC;
             EN : in STD_LOGIC;
           CLR : in STD_LOGIC;
              D : in STD_LOGIC_vector(1 downto 0);
              Q : out STD_LOGIC_vector(1 downto 0));
end component;
component RMUX2_2 is
    port (MA : in std_logic_vector(1 downto 0);
```

```
            MB : in std_logic_ vector (1 downto 0);
           CLK : in std_logic;
             S : in std_logic_ vector (0 downto 0);
             Q : out std_logic_vector (1 downto 0));
end component;
component AND2I1 is
   Port ( X : in std_logic;
          Y : in std_logic;
          Z : out std_logic);
end component;
component RMUX2_64 is
    port (  MA : in std_logic_vector (63 downto 0) ;
            MB : in std_logic_vector (63 downto 0) ;
             S : in std_logic_ vector (0 downto 0) ;
           CLK : in std_logic;
             O : out std_logic_ vector (63 downto 0) ) ;
end component;
signal N1:std_logic;
signal N2:std_logic_vector(63 downto 0);
signal N3:std_logic_vector(63 downto 0);
signal N4:std_logic;
signal N5:std_logic_vector(63 downto 0);
signal N6:std_logic_vector(63 downto 0);
signal N7:std_logic_vector(64 downto 0);
signal N8:std_logic;
signal N9:std_logic_vector(1 downto 0);
signal N10:std_logic_vector(63 downto 0);
signal N11:std_logic_vector(64 downto 0);
signal N12:std_logic_vector(65 downto 0);
signal N13:std_logic_vector(63 downto 0);
signal N14:std_logic_vector(1 downto 0);
signal N15:std_logic_vector(1 downto 0);
signal N16:std_logic_vector(63 downto 0);
signal N17:std_logic;
```

```
signal N18:std_logic_vector(63 downto 0);
signal N19:std_logic;
signal GND:std_logic;
signal GND2:std_logic_vector(1 downto 0);
signal GND62:std_logic_vector(61 downto 0);
begin
GND<='0';
GND2 <="00";
GND62<="00000000000000000000000000000000000000000000000000000000000000";
U1:AND2I1 port map(X=>K2,Y=>H4,Z=>N1);
U2:ram128_64 port map(clk=>Hb,en=>Lc,we=>N1,din=>N18,addr=>Z,dout=>N2);
U3:RLatch64 port map(clk=>H3,Reset=>RST,D=>N2,Q=>N3);
U4:rsadd64 port map(A=>PI, B=>N3, ACLR=>RST, clk=>H4, C_in=>GND,
Q_C_OUT=>N4, Q=>N5);
U5:rmux2_64 port map(MA=>N2,MB=>N16,S(0)=>K1,Q=>N6,CLK=>H4);
U6:inout1_64s65 port map(xin=>N6,yin=>GND,zout=>N7);
U7:OR6 port map(E=>X,S=>N8);
U8:rmux2_2 port map(MA=>GND2,MB=>N14,S(0)=>N8,Q=>N9,clk=>H3);
U9:inout2_62s64 port map(xin=>N9,yin=>GND62,zout=>N10);
U10:radd64 port map(A=>PS,B=>N10,SCLR=>RST,C_in=>GND,clk=>H4,Q=>N11);
U11:radd65 port map(A=>N7,B=>N11,SCLR=>RST,C_in=>N4,clk=>H1,Q=>N12);
U12:inout66s2_64 port map(xin=>N12,yout=>N13,zout=>N14);
U13:RELatch2 port map(clk=>H2,Reset=>RST,EN=>K1,D=>N14,Q=>N15);
U14:inout2_62s64 port map(xin=>N15,yin=>GND62,zout=>N16);
U15:AND1 port map(X=>H1,Y=>H4,Z=>N17);
U16:mux2_64 port map(MA=>N5,MB=>N13,S(0)=>N17,O=>N18);
U17:AND2I1 port map(X=>K4,Y=>K3,Z=>N19);
U18:TRS64 port map(X=>N2,T=>N19,Y=>R);
end Structural;
```

----La description en langage VHDL de la partie opérative se fait de la manière suivante :

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

```
use ieee.std_logic_unisigned.all;
entity POP is
   Port ( RST, Hb, H1, H2, H3, H4, LC, K2, K3: in std_logic ;
    X : out std_logic_vector(5 downto 0) ;
Y : out std_logic_vector(5 downto 0);
 Z : out std_logic_vector(6 downto 0);
R : out std_logic_vector(63 downto 0));
end POP;
architecture Structural of POP is
component MADR is
    Port ( RST : in std_logic;
                 H : in std_logic;
                H1 : in std_logic;
                H2 : in std_logic;
                Lc : in std_logic;
               K1 : in std_logic;
               K2 : in std_logic;
               K4 : in std_logic;
                 X : out std_logic_vector(5 downto 0);
                 Y : out std_logic_vector(5 downto 0);
                 Z : out std_logic_vector(6 downto 0));
end component;
component MMUL is
  Port ( RST : in std_logic;
               Lc : in std_logic;
              H2 : in std_logic;
              H3 : in std_logic;
                X : in std_logic_vector(5 downto 0);
                Y : in std_logic_vector(5 downto 0);
               PI : out std_logic_vector(63 downto 0);
              PS : out std_logic_vector(63 downto 0));
end component;
component MACC is
Port ( RST : in std_logic;
```

```
        Hb : in std_logic;
        H1 : in std_logic;
        H2 : in std_logic;
        H3 : in std_logic;
        H4 : in std_logic;
        LC : in std_logic;
        K1 : in std_logic;
        K2 : in std_logic;
        K3 : in std_logic;
        K4 : in std_logic;
         X : in std_logic_vector(5 downto 0);
         Z : in std_logic_vector(6 downto 0);
        PI : in std_logic_vector(63 downto 0);
        PS : in std_logic_vector(63 downto 0);
         R : out std_logic_vector(63 downto 0));
end component;
signal N1: std_logic_vector (5 downto 0) ;
signal N2: std_logic_vector (5 downto 0) ;
signal N3: std_logic_vector (6 downto 0);
signal N4: std_logic_vector (63 downto 0);
signal N5: std_logic_vector (63 downto 0);
begin
U1: MADR port map (RST=>RST, Lc=>Lc, H=>H, H1=>H1 ,H2=>H2, K1=>K1,
K2=>K2, K4=>K4, X=>N1,Y=>N2, Z=>N3);
U2: MMUL port map(RST=>RST, LC=>LC,H2=>H2,H3=>H3, X=>N1,Y=>N2, PI=>N4,
PS=>N5);
U3: MACC port map(RST=>RST, Hb=>Hb, H1=>H1, H2=>H2, H3=>H3, H4=>H4,
Lc=>Lc, K1=>K1, K2=>K2, K3=>K3, K4=>K4, X=>N1,Z=>N3, PI=>N4, PS=>N5,
R=>R) ;       .
X<=N1;
Y<=N2;
Z<=N3;
end Structural;
```

----La description en langage VHDL de module générateur des signaux d'horloge se fait de la manière suivante :

```
library ieee;
library unisim;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
useunisim.vcomponents.all;
entity MGSH is
   port ( clk_in : in     std_logic;
        clk1_out : out std_logic;
        clk2_out : out std_logic;
        clk3_out : outstd_logic;
        clk4_out : out std_logic;
        clk5_out : out std_logic;
        clk6_out : out std_logic);
        clk7_out : out std_logic;
end MGSH;
architecture schematic of  MGSH is
SIGNAL XLXN_1: STD_LOGIC;
SIGNAL XLXN_10: STD_LOGIC;
SIGNAL XLXN_11: STD_LOGIC;
SIGNAL XLXN_12: STD_LOGIC;
SIGNAL XLXN_13: STD_LOGIC;
SIGNAL XLXN_15: STD_LOGIC;
SIGNAL XLXN_16: STD_LOGIC;
SIGNAL XLXN_18: STD_LOGIC;
SIGNAL XLXN_2: STD_LOGIC;
SIGNAL XLXN_20: STD_LOGIC;
SIGNAL XLXN_21: STD_LOGIC;
SIGNAL XLXN_22: STD_LOGIC;
SIGNAL XLXN_28: STD_LOGIC;
SIGNAL XLXN_29: STD_LOGIC;
SIGNAL XLXN_30: STD_LOGIC;
SIGNAL XLXN_4: STD_LOGIC;
```

```
SIGNAL XLXN_6: STD_LOGI
SIGNAL XLXN_7: STD_LOGIC;
SIGNAL XLXN_9: STD_LOGIC;
SIGNAL clk1_out_DUMMY: STD_LOGIC;
SIGNAL clk3_out_DUMMY: STD_LOGIC;
ATTRIBUTE fpga_dont_touch : STRING ;
ATTRIBUTE fpga_dont_touch OF XLXI_18 : LABEL IS "true";
ATTRIBUTE fpga_dont_touch OF XLXI_17 : LABEL IS "true";
ATTRIBUTE fpga_dont_touch OF XLXI_4 : LABEL IS "true";
ATTRIBUTE fpga_dont_touch OF XLXI_5 : LABEL IS "true";
ATTRIBUTE fpga_dont_touch OF XLXI_6 : LABEL IS "true";
ATTRIBUTE fpga_dont_touch OF XLXI_7 : LABEL IS "true";
ATTRIBUTE fpga_dont_touch OF XLXI_8 : LABEL IS "true";
ATTRIBUTE fpga_dont_touch OF XLXI_9 : LABEL IS "true";
ATTRIBUTE fpga_dont_touch OF XLXI_10 : LABEL IS "true";
ATTRIBUTE fpga_dont_touch OF XLXI_11 : LABEL IS "true";
ATTRIBUTE fpga_dont_touch OF XLXI_12 : LABEL IS "true";
ATTRIBUTE CLKDV_DIVIDE : STRING ;
ATTRIBUTE DUTY_CYCLE_CORRECTION : STRING ;
ATTRIBUTE fpga_dont_touch OF XLXI_14 : LABEL IS "true";
ATTRIBUTE CLKDV_DIVIDE OF XLXI_14 : LABEL IS "2";
ATTRIBUTE DUTY_CYCLE_CORRECTION OF XLXI_14 : LABEL IS "TRUE";
ATTRIBUTE fpga_dont_touch OF XLXI_2 : LABEL IS "true";
ATTRIBUTE CLKDV_DIVIDE OF XLXI_2 : LABEL IS "2";
ATTRIBUTE DUTY_CYCLE_CORRECTION OF XLXI_2 : LABEL IS "TRUE";
ATTRIBUTE fpga_dont_touch OF XLXI_1 : LABEL IS "true";
ATTRIBUTE CLKDV_DIVIDE OF XLXI_1 : LABEL IS "2";
ATTRIBUTE DUTY_CYCLE_CORRECTION OF XLXI_1 : LABEL IS "TRUE";
ATTRIBUTE INIT : STRING ;
ATTRIBUTE fpga_dont_touch OF XLXI_19 : LABEL IS "true";
ATTRIBUTE INIT OF XLXI_19 : LABEL IS "0";
ATTRIBUTE fpga_dont_touch OF XLXI_13 : LABEL IS "true";
ATTRIBUTE fpga_dont_touch OF XLXI_15 : LABEL IS "true";
ATTRIBUTE fpga_dont_touch OF XLXI_16 : LABEL IS "true";
```

```
ATTRIBUTE IOSTANDARD : STRING ;
ATTRIBUTE fpga_dont_touch OF XLXI_3 : LABEL IS "true";
ATTRIBUTE IOSTANDARD OF XLXI_3 : LABEL IS "LVTTL";
BEGIN
  clk1_out <= clk1_out_DUMMY;
  clk3_out <= clk3_out_DUMMY;
  XLXI_18 : BUFG
    PORT MAP (I=>XLXN_9, O=>XLXN_7);
  XLXI_17 : BUFG
    PORT MAP (I=>XLXN_28, O=>XLXN_29);
  XLXI_4 : BUFG
    PORT MAP (I=>XLXN_2, O=>clk1_out_DUMMY);
  XLXI_5 : BUFG
    PORT MAP (I=>XLXN_4, O=>clk2_out);
  XLXI_6 : BUFG
    PORT MAP (I=>XLXN_6, O=>XLXN_28);
  XLXI_7 : BUFG
    PORT MAP (I=>XLXN_18, O=>XLXN_16);
  XLXI_8 : BUFG
    PORT MAP (I=>XLXN_20, O=>XLXN_9);
  XLXI_9 : BUFG
    PORT MAP (I=>XLXN_10, O=>clk3_out_DUMMY);
  XLXI_10 : BUFG
    PORT MAP (I=>XLXN_11, O=>clk4_out);
  XLXI_11 : BUFG
    PORT MAP (I=>XLXN_12, O=>clk5_out);
  XLXI_12 : BUFG
    PORT MAP (I=>XLXN_13, O=>clk6_out);
  XLXI_14 : CLKDLL
    PORT MAP (CLKFB=>XLXN_16, CLKIN=>XLXN_29, RST=>XLXN_21,
CLK0=>XLXN_18,
    CLK180=>open, CLK270=>open, CLK2X=>open, CLK90=>open,
CLKDV=>XLXN_20,
    LOCKED=>open);
```

```
  XLXI_2 : CLKDLL
    PORT MAP (CLKFB=>clk3_out_DUMMY, CLKIN=>XLXN_7, RST=>XLXN_22,
    CLK0=>XLXN_10, CLK180=>XLXN_12, CLK270=>XLXN_13, CLK2X=>open,
    CLK90=>XLXN_11, CLKDV=>open, LOCKED=>XLXN_30);
  XLXI_1 : CLKDLL
    PORT MAP (CLKFB=>clk1_out_DUMMY, CLKIN=>XLXN_1, RST=>XLXN_15,
    CLK0=>XLXN_2, CLK180=>XLXN_4, CLK270=>open, CLK2X=>open,
CLK90=>open,
    CLKDV=>XLXN_6, LOCKED=>open);
  XLXI_19 : FD
    PORT MAP (C=>XLXN_2, D=>XLXN_30, Q=>D_out);
  XLXI_13 : GND
    PORT MAP (G=>XLXN_15);
  XLXI_15 : GND
    PORT MAP (G=>XLXN_21);
  XLXI_16 : GND
    PORT MAP (G=>XLXN_22);
  XLXI_3 : IBUFG
    PORT MAP (I=>clk_in, O=>XLXN_1);
end schematic;
```

----La description en langage VHDL du module générateur des signaux de commande se fait de la manière suivante :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unisigned.all;
entity MGSC is
   Port ( RST : in std_logic;
             L : in std_logic_vector(5 downto 0);
          A1 : in std_logic;
          A2 : in std_logic;
          A3 : in std_logic;
           X : in std_logic_vector(5 downto 0);
```

```
            Y : in std_logic_vector(5 downto 0);
            Z : in std_logic_vector(6 downto 0);
K1 : out std_logic;
K2 : out std_logic;
K3 : out std_logic;
K4 : out std_logic;
end mgsc;
architecture Structural of MGSC is
component rcmp2_6
   port ( A : in std_logic_vector(5 downto 0);
          B : in std_logic_vector(5 downto 0);
      CLK : in std_logic;
    ACLR : in std_logic;
 QA_EQ_B : out std_logic);
end component;
component rcmp2_7
   port ( A : in std_logic_vector(6 downto 0);
          B : in std_logic_vector(6 downto 0);
       CLK : in std_logic;
        CE : in std_logic;
QA_EQ_B : out std_logic);
end component;
component add6
   port ( A : in std_logic_vector(5 downto 0);
          B : in std_logic_vector(5 downto 0);
      C_IN : in std_logic;
           S : out std_logic_vector(6 downto 0));
end component;
component RLatch1 is
   Port ( clk : in std_logic;
            D : in std_logic;
        Reset : in std_logic;
            Q : out std_logic);
end component;
```

```
component RDLatch1 is
  Port ( clk : in std_logic;
               D : in std_logic;
         Reset : in std_logic;
             Q : out std_logic);
end component;
signal N1: std_logic ;
signal N2: std_logic;
signal N3: std_logic_vector(6 downto 0);
signal N4: std_logic;
signal N5: std_logic;--Signal intermediare
signal VCC: std_logic;
begin
VCC<='1';
U1: rcmp2_6 port map(A=>X,B=>L, clk=>A3, ACLR=>RST, QA_EQ_B=>K1);
U2: rcmp2_6 port map(A=>Y,B=>L, clk=>A2,ACLR=>RST,QA_EQ_B=>N1);
 U3: RDLatch1 port map(clk=>N1, Reset=>RST, D=>VCC,Q=>N2);
U4:RLatch1 port map(clk=>A1,Reset=>RST,D=>N2,Q=>K3);
U5: add6 port map(A=>L,B=>L, C_IN=>VCC,S=>N3);
U6: rcmp2_7 port map(A=>Z,B=>N3,clk=>A1,CE=>N2,QA_EQ_B=>N4);
U7:RLatch1 port map(clk=>N4,Reset=>RST,D=>VCC,Q=>N5);
U8:RLatch1 port map(clk=>A1,Reset=>RST,D=>N5,Q=>K4);
 K2<=N2;
end Structural;
```

----La description en langage VHDL de l'architecture globale se fait de la manière suivante :

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unisigned.all;
entity MULVP is
    Port ( CLK : in std_logic;
               RST : in std_logic;
```

```
            L : in std_logic_vector (5 downto 0);
            R : out std_logic_vector (63 downto 0));
end MULVP;
architecture structural of MULVP is
component PCT is
  Port ( CLK, RST : in std_logic;
            L : in std_logic_vector(5 downto 0);
           X : in std_logic_vector(5 downto 0);
           Y : in std_logic_vector(5 downto 0);
           Z : in std_logic_vector(6 downto 0);
           H, Hb, H1, H2, H3, H4, Lc, K1, K2, K3, K4 : out std_logic);
end component;
component POP is
   Port ( RST, H, Hb, Lc, H1, H2, H3, H4, K1, K2, K3, , K4 : in std_logic;
          X : out std_logic_vector (5 downto 0);
          Y : out std_logic_vector (5 downto 0);
          Z : out std_logic_vector (6 downto 0);
          R : out std_logic_vector (63 downto 0) ) ;
end component;
signal N1: std_logic;
signal N2: std_logic;
signal N3: std_logic;
signal N4: std_logic;
signal N5: std_logic;
signal N6: std_logic;
signal N7: std_logic;
signal N8: std_logic;
signal N9: std_logic;
signal N10: std_logic;
signal N11: std_logic;
signal N12: std_logic_vector (5 downto 0) ;
signal N13: std_logic_vector (5 downto 0);
signal N14: std_logic_vector (6 downto 0);
begin
```

U1: PCT port map (CLK=>CLK, RST=>RST, L=>L, Lc=>N7, X=>N12, Y=>N13, Z=>N14, H=>N1, Hb=>N2, H1=>N3, H2=>N4, H3=>N5, H4=>N6, K_1=>N8, K_2=>N9, K_3=>N10, K_4=>N11) ;

U2: POP port map (RST=>RST, H=>N1, Hb=>N2, H1=>N3, H2=>N4, H3=>N5, H4=>N6, Lc=>N7, K_1=>N8, K_2=>N9, K_3=>N10, K_4=>N11, X=>N12, Y=>N13, Z=>N14, R=>R); end Structural

APPENDICE B
CALCUL PAR MAPLE 6

Afin de vérifier le résultat obtenu lors de la simulation de l'architecture globale, on à effectué le calcul R=A×B en utilisant l'outil MAPLE 6 :

Restart ; P : =
(‘ 987654FEDCBA1FED987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321
987654FEDCBA1FED987654FEDCBACBA9987654FEDCBA8765987654FEDCBA432198
7654FEDCBA1FED987654FEDCBACBA9987654FEDCBA8765987654FEDCBA43219876
54FEDCBA1FED987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321987654
FEDCBA1FED987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321987654F
EDCBA1FED987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321987654FE
DCBA1FED987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321987654FED
CBA1FED987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321987654FEDC
BA1FED987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321987654FEDCB
A1FED987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321987654FEDCBA
1FED987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321987654FEDCBA1F
ED987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321987654FEDCBA1FE
D987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321987654FEDCBA1FED
987654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321987654FEDCBA1FED98
7654FEDCBACBA9987654FEDCBA8765987654FEDCBA4321987654FEDCBA1FED9876
54FEDCBACBA9987654FEDCBA8765987654FEDCBA4321’ , décimal, hex)*convert
(‘A987654321FEDCBA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A
987654321FEDCBA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A987
654321FEDCBA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A987654
321FEDCBA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A987654321
FEDCBA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A987654321FE
DCBA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A987654321FEDC
BA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A987654321FEDCBA
987654321FEDCBA987654321FEDCBA9876543FEDCBA98765

A987654321FEDCBA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A98
7654321FEDCBA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A98765
4321FEDCBA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A98765432
1FEDCBA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A987654321FE
DCBA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A987654321FEDC
BA987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A987654321FEDCBA
987654321FEDCBA987654321FEDCBA9876543FEDCBA98765A987654321FEDCBA987
654321FEDCBA987654321FEDCBA9876543FEDCBA98765', décimal, hex) ; R : = convert
(P, hex) ;

R :=64F6C0D1BCDD1AD5E4651DA5783AD8D7CB5D1FE362AA2D01300911DEC832E9
9673C0A08233F5192565DE6B8F325DCCE5BC5276F9DD959EDA693BFEDA3B2403769
81533E89A1884EC856AFCC6E10E0C4EC2D2EFBD90886DA1126EEBD5AE151D56BC6
9C74F003BF0B3A4F78DFE8FBE4BB7C9536881437B3C67BBA1D8D121063736E0BE5A
B5665F5C7AC481F363E6E8B20CFD3E144F66E07B2E64D4C5CC93F751170512EE1BCC
82C841E410B06DED1ECA89D6545A08A960D9F50E07B2C806E86AF72967818232A634
09039D41A59BCF09F2DCD4D2CC5C53A8BBB73A9FC379D984D74DBC14E898C99FD
02329D2DD4A7F495BE3554B900F467782606D8CBEECCA9EB77210A84EFEEED0B974
2B66414F92F88C4E9D5C453C23946909A079BA079BA5FBB89796653BB56510775E624
2F4CA7DFC82DF0563D17752C150FB2D366B5D2ACD277BAB9CF1BCA15C17BC0540
46FFFD57572E96B5DB281EE3D65C0653B145DEC57DF0E628231574EECA15C17BC05
4046FFFD572E9EDB11B2940ACB88F06380362F5E8977ABAB3C0E8A8F3F0866903D7A
7628E048163AE6C2DA82B80201827B7894EFD9E267AE053A2B62A0C55D20A5820264
0F7502A579F1AA39E7139F84C0C1CB563C192420001CB631141053B10D898E9F3EA1E
F100D2079F116253D87060B01BC9E4FE091F8E5C9AC00144A4175911ADA6DCEDB7A
A04F49A84536DABF5DD294A94650F13B277392367566913B2358E05048A1C56D325B
DD866DEB46BD18895E42E41A34C92EAA7EBB8A629AAABB5154151974D9029F2D0
DCF3ED9A8BAD34027DC81E3785735E16B409F5EA1A5349C7D9EA47EC980BFE3E18
0FC80AB88DF6F1761FACBBE53D187C5B45AAD89EB423B9EB1B002D78CF6E60E0B6
6CB648ADBB0FBD75FF73444C956B6F9833AAF638BEE13C2E5A09EA9B1A4CEB4036
84A95B3F43014B8630CFDE52C0691B3323285CC12758271CEF2829330B1BE1BE42F90
8868F52BD1D54F34EC9D8B32B96ED2DD943AEDBF42FF3B538192AF78D217DC96D1
CA1D59F409DA084AD3484B240AB1FE674E832C14246FF4AD3388E176349B0DAB612
AF65F1D46DCB7E31C7E7621898E55FCCF55E56AAEE3FAB0FD26451396861E2E43242
E6277B14F23BA00D75AE05B69015C64CB8A8F674A9702D94C76F98CF69473EEF9713

0D660070E089437287C05E973B1AA663AD3E430DF6D85576CFBBEE5C3AF906AB05D
9C541B39CD8DF58F0A768297CA41E6407DCABC01BE37E06A8737137027642FE4D8A
1286B06E4C2A20F03917DEA1D91AD17649895F71853D94F886330BE5DAF6C14EE359
C403B8FB51390A06409F93D19B0FE752A2FF8C74817D9B2F155572EF3513B42CD7992
5CC8181DAF4A29D4E8864A9851BE68D93AB6E02B02HB1EC50AE7A8D884FFEB2E92
9DB1CAABE305.

APPENDICE C

LISTE DES ABREVIATIONS

ASIC : **A**pplication **S**pecific **I**ntegrated **C**ircuit

CLB : **C**onfigurable **L**ogic **B**lock

DSP : **D**igital **S**ignal **P**rocessing

DCM : **D**igital **C**lock **M**anager

DLL : **D**elay **L**ocked **L**oop

FPGA : **F**ield **P**rogrammable **G**ate **A**rray

GRM : **G**eneral **R**outing **M**atrix

GCM : **G**lobal **C**lock **M**ux

IOB : **I**nput/**O**utput **B**lock

IEEE : **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers

ISE : **I**ntegrated **S**oftware **E**ngeneering

LUT : **L**oock **U**p **T**able

PLD : **P**rogrammable **L**ogic **D**evice

SRAM : Bloc **S**elect **RAM**

VLSI :**V**ery **L**arge **S**cale **I**ntegration

VHDL : **V**ery **H**igh-speed integrated circuits **H**ardware **D**escription **L**anguage

XC : **X**ilinx **C**ircuit

REFERENCES

1. J.M.Muller “Arithmétique des ordinateurs” Masson, Paris 1989.

2. M.Daumas et J.M.Muller “Qualité des calcules sur ordinateur”, Edition Masson, Paris, 1997.

3. M.J.Schulte “A variable precision, interval arithmetic processor ”, the university of texas at austin, may1996.

4. R.P.Brent, “A Fortran Multiprecision Arithmetic Package, ”ACM Transaction on Mathematical Software”, vol.4, pp.57-70, 1978.

5. R.P. Brent, “Multiple-Precision Zero Finding Methods and the Complexity of Elementary Function Evaluation”, in Analytic Computational Complexity Function Evaluation, ed, Academic Press, pp.151-176, 1976.

6. D.H.Bail, “Algorithm 719 Multiprecision Translation and Execution of Fortran Programs ”, ACM Transaction Mathematical Software, vol.19 pp.288-319, 1993.

7. M .S.Cohen, T. E.Hull, and V. C. Hamacher, CADAC: “A Conrolled-Precision Decimal Arithmetic Unit, IEEE Transactions on computes ”, vol. C-32, pp. 370-377, 1983.

8. “Introduction au micro processeur et circuits assimilés”, cinquième partie(5/5) : Les composants FPGA, sur site Internet : ttp://perso.wanadoo.fr/michel.hubin/physique /microp/chap_mp5.htm.

9. “Virtex II Platform FPGAs: Introduction and Overview”, DS031-1(V1.9) September 26, 2002, Pages (2, 3, 6).

10. “Virtex II 1.5 Field Programmable Gate Arrays”, module 2. DS031-2 (V1.9). November 29, 2001, Page (1, 2, 12, 13, 17, 21, 26, 30, 33).

11. “Dual-Port Bloc Memory for Virtex-E, Virtex-II, Virtex Pro, Spartan II et Spartan IIE”, V4.0. October 4, 2001.

12. “Virtex II1.5 Field Programmable Gate Arrays Module 3”, DS031-3 ,(V2.0). November 28, 2001. Page (6.18)

13. Xilinx, “Synthesis and Simulation Design Guide”, Xinx, INC, USA-1998.