12/04

Ministère de l'Enseignement Supérieur et de
la Recherche Scientifique

ECOLE NATIONALE
POLYTECHNIQUE

وزارة التعليـــــــم العالـــــــي
و البحث العلمـــــــي
المدرسة الوطنية المتعددة التقنيات

DÉPARTEMENT D'ÉLECTRONIQUE

IMPLÉMENTATION D'ALGORITHMES DU PROBLÈME DE CHEMIN ALGÉBRIQUE

PROJET DE FIN D'ÉTUDES PRÉSENTÉ EN VUE
DE L'OBTENTION DU DIPLÔME
D'INGÉNIEUR D'ÉTAT EN ÉLECTRONIQUE

Etudié par
Mr METREF Adel
Mr GACEM Youcef

Proposé par
A.K OUDJIDA

Encadré par
R.SADOUN
A.K OUDJIDA

Juin 2004

Je dédie ce travail à mes chers parents et grands-parents, mes frères, toute ma famille et tous mes amis.
Youcef.

Je dédie ce travail à mes très chers parents, à mes chers amis Youcef, Atef, Abderrazak, M'hand et à tous ceux qui me connaissent.
Adel.

REMERCIEMENTS

Nous tenons avant tout à remercier notre professeur R. SADOUN pour le soutien qu'il nous a apporté lors de la préparation de ce rapport. Par ailleurs, un grand merci à Monsieur A. K. OUDJIDA pour nous avoir proposé le sujet et pour les conseils avisés qu'il a su nous donner tout au long de notre travail. Merci aussi à tous ceux, camarades et professeurs, qui ont su nous accorder une partie de leur temps précieux lors de la préparation de ce projet.

TABLE DES MATIÈRES

LISTE DES ABRÉVIATIONS, DES SIGLES ET DES ACRONYMES

ASIC: Application Specific Integrated Circuit
CAO: Conception Assistée par Ordinateur
CPLD: Complex Programmable Logic Device
cfc: Composante fortement connexe
cfcm: Composante fortement connexe maximale
CLB: Configurable Logic Box
DCM: Digital Clock Managment
DOD: Departement of Defense
DSP: Digital Signal Processor
EEPLD: Electrically Erasable Programmable Logical Device
EP: Elementary Processor
EPLD: Erasable PLD
FPGA: Field Programmable Gate Array
GCLK: Global Clock
LP: Liste des prédecesseurs
LS: Liste des successeurs
LSB: Least Significant Bit
LSI: Large Scale Integration
LUT: Look Up Table
MOS: Metal Oxide Semiconductor
MSB: Most Significant Bit
MSI: Medium Scale Integration
MWST: Minimum Weight Spanning Tree
PAL: Programmable Array Logic
PLA: Programmable Logic Array
PLD: Programmable Logic Device
PSI ψ: Problem Size Independence
ROM: Read Only Memory
SP: Shortest Path
SRAM: Static Random Access Memory
SSI: Small Scale Integration
TC: Transitive Closer
VHSIC: Very High Speed Integrated Circuit
VHDL: VHSIC Hardware Description Language
VLSI: Very Large Scale Integration

LISTE DES FIGURES

ملخص:
يتناول هذا العمل دراسة ثلاث خوارزميات لمعضلة المسار الجبري وهي: الإغلاق المتعدّ لبيان، طول اقصر مسار في بيان و الجدع الشامل لبيان ذو أدنى مقدار.
تم الحصول على بنية انقباضية مستوحاة من خوارزمية Warshall-Floyd (الأحسن من ناحية سرعة التنفيذ)، و كذا إدماجها في الدّارة المنطقية المبرمجة FPGA عائلة Virtex-II و اختبارها على المستوى الو ضيفي و على المستوى post-place and route من أجل قيم مختلفة لمعطيات المشكل.
الكلمات المفتاحية: معضلة المسار الجبري, البنى الانقباضية, خوارزمية Warshall-Floyd, FPGA

Résumé :
Ce travail traite trois algorithmes du problème du chemin algébrique a savoir : la fermeture transitive d'un graphe, la longueur du plus court chemin dans un graphe et l'arbre couvrant de poids minimum.
Une architecture systolique inspirée de l'algorithme de Warshall-Floyd a été élaborée, implémentée pour un circuit FPGA de la famille Virtex-II et testée au niveau fonctionnel et au niveau post-placement et routage pour différentes dimensions du problème. Le temps d'exécution de cette architecture est N cycles d'horloge, N étant la taille de la matrice qui représente le graphe.
Mots Clefs: Problème du chemin algébrique, Architectures systoliques, Algorithme de Warshall-Floyd, FPGA

Abstract :
This work deals with three algorithms of the algebraic path problem: the transitive closure, the shortest path and the minimum weight spanning tree.
A systolic architecture inspired from the Warshall-Floyd algorithm (the most optimal in term of time execution) have been developed, implemented on Xilinx's FPGA using the Virtex-II family and tested at the functional level and the post-place and route level for different input data of the problem.
Key words: Algebraic path problem, Systolic array, Warshall-Floyd Algorithm, FPGA

INTRODUCTION

Le concept de chemin algébrique formalise et unifie des algorithmes qui, a priori, n'ont rien de commun. Ces algorithmes font partie de la classe des algorithmes d'optimisation. Ils trouvent leur application dans des domaines variés. On cite la recherche opérationnelle, la logistique, le routage de circuits électroniques, la conception de réseaux informatiques, le transport d'énergie électrique, l'intelligence artificielle voire la robotique. Ces algorithmes sont la fermeture transitive d'un graphe, la longueur du chemin le plus court entre deux sommets d'un graphe et l'arbre couvrant de poids minimum.

L'exécution de ces algorithmes s'apparente à la multiplication de matrices N×N; ce qui peut prendre un temps considérable ($N^3\log_2(N)$) [1] si on les exécute séquentiellement.

Cependant, les progrès importants fait en matière de circuits VLSI ces deux dernières décennies et l'avènement des architectures systoliques en 1978 permettent d'envisager une autre façon de faire.

Un algorithme séquentiel [1] adapaté aux architectures systoliques dit de Warshall-Floyd réduit le nombre d'opérations à N^3.

Jusqu'à présent et à notre connaissance, les meilleurs résultats ont été obtenus par Tsay et Co [1] avec un temps d'exécution de 4N-2.

Il s'agit dans notre présent travail de concevoir et d'implémenter sur des circuits FPGA de Xilinx une architecture qui permet de résoudre ces problèmes en N cycles d'horloge.

Pour mettre en valeur notre solution et proposer une justification de notre approche, nous avons organisé notre mémoire en six chapitres.

Le premier fait une introduction à la théorie des graphes. Il aborde quelques notions nécessaires pour la compréhension des développements que nous avons introduits.

Le deuxième chapitre aborde l'approche théorique du problème de chemin algébrique; les algorithmes y seront démontrés et expliqués. Le troisième chapitre détaille les architectures systoliques. Le quatrième chapitre constitue un rappel de ce qu'est un FPGA et de ce qu'est le langage VHDL. Il met aussi en évidence les spécificités de la famille Virtex II qui sera notre cible technologique.

Le cinquième chapitre aborde les détails des architectures proposées en partant de la solution algorithmique. A partir de cette dernière, on évalue les besoins en

---

[1] N est la taille de la matrice qui représente le graphe

matière de circuits et au fur et à mesure que les contraintes d'implémentation apparaissent, on apporte des solutions pour aboutir enfin à l'architecture finale. Le sixième chapitre donne les détails de l'implémentation et des tests. Il présente aussi un testbench automatique qu'on a développé afin de tester l'architecture au niveau comportementale. Ce test peut être étendu au niveau du post-placement et du routage.

CHAPITRE I

INTRODUCTION A LA THEORIE DES GRAPHES[1]

La théorie des graphes est née en 1736 quand Euler démontra qu'il était impossible de traverser chacun des sept ponts de la ville russe de Königsberg (aujourd'hui Kaliningrad) une fois exactement et de revenir au point de départ. Les ponts enjambent les bras de la Pregel qui coulent de part et d'autre de l'île de Kneiphof. Dans la figure suivante, les noeuds représentent les rives.

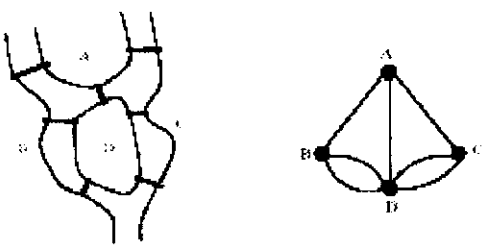


Figure 1.1 : Schéma représentant les 7 ponts de Kaliningrad.

Elle s'est surtout développée depuis la deuxième moitié du 19ème siècle (avec Hamilton, Heawood, Kemp, Kirchhoff, Petersen, Tait), et connaît un grand boom depuis les années 30 (avec König, Hall, Kuratowski, Whitney, Erdös, Tutte, Edmonds, Berge, Lovász, Seymour, et beaucoup d'autres). Elle présente des liens évidents avec l'algèbre, la topologie, et d'autres domaines de la combinatoire. On trouve des applications de la théorie des graphes – et souvent aussi la motivation de nouveaux problèmes – en informatique, recherche opérationnelle, théorie des jeux, théorie de la décision.

1.1 DÉFINITION D'UN GRAPHE

Un graphe est une représentation symbolique d'un réseau. Il s'agit d'une abstraction de la réalité de sorte à permettre sa modélisation. En géographie des transports, la plupart des réseaux ont un fondement spatial, mais ceci n'est pas vrai pour tous les réseaux de transport. Par exemple, il est possible de représenter un système de télécommunication sous forme de réseau bien que son expression spatiale a une importance limitée et serait difficile à transposer sur un territoire. Les exemples

[1] Ce chapitre a été rédigé en s'appuyant sur la référence [2]

d'un réseau de téléphones mobiles ou l'Internet incarnent des cas de réseaux à structure spatiale limitée. Toutefois, la majorité des réseaux de transport peuvent être représentés par le biais de la théorie des graphes.

1.1.1 GRAPHE ORIENTÉ, NON ORIENTÉ

Un graphe peut être *orienté* ou *non orienté.*

Un graphe orienté peut représenter par exemple la relation de filiation dans un arbre généalogique, si A est le père de B, alors B ne peut être le père de A. Un graphe

G(X, U) peut être déterminé par :

- $X = \{x_1, x_2, x_3, \ldots, x_n\}$ dont les éléments sont appelés sommets ou nœuds ;
- Un ensemble $U = \{u_1, u_2, \ldots, u_m\}$ du produit cartésien $X \times X$ dont les éléments sont appelés arcs.

Pour un arc (figure 1.2) $u = (x_i, x_j)$, x_i est l'extrémité initiale, x_j l'extrémité terminale (ou bien origine et destination). L'arc u part de x_i et arrive à x_j.

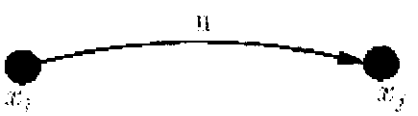


Figure 1.2: Un arc

Figure 1.3: Une boucle

Un arc u= (xi , xi) est appelé une boucle (figure 1.3).

Un graphe non orienté (figure 1.4) peut représenter par exemple une relation de conflit entre objet ou individu, car dire que A est en conflit avec B revient au même que dire B est en conflit avec A. Un arc non orienté est appelé une arrête, U est constitué non pas de couples mais de paires de sommets non ordonnés. Dans ce cas on note G(X, E) au lieu de G(X, U), e= [xi, xj] et on dit que e est incident à xi et xj.

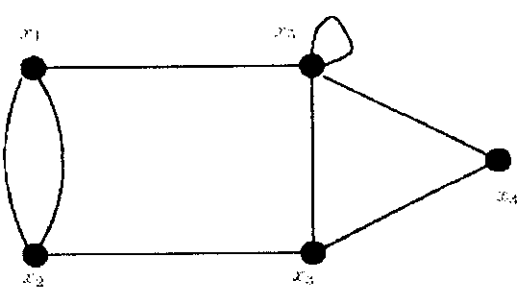
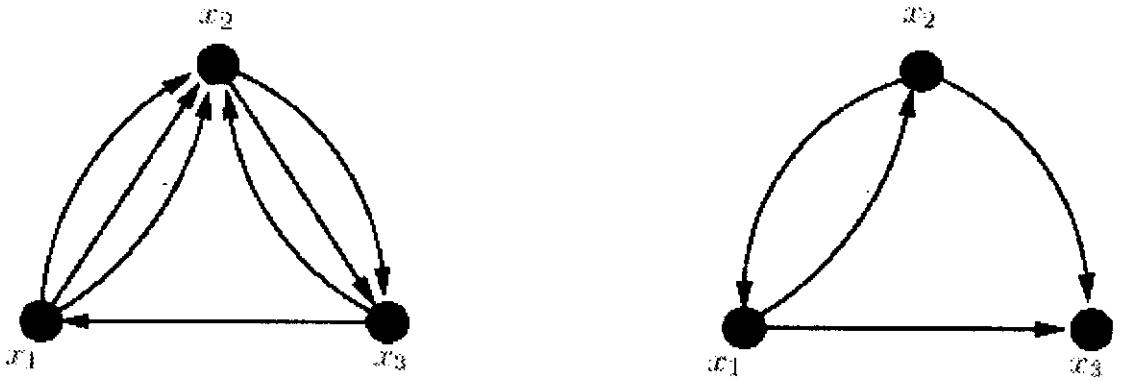


Figure 1.4 : Exemple de graphe non orienté

Un p-graphe (figure 1.5) est un graphe dans lequel il n'existe jamais plus de p arcs de la forme (i, j) entre deux sommets quelconques.



Un 3-graphe Un 1-graphe ≡ graphe

Figure 1.5

1.1.2 APPLICATION MULTIVOQUE

x_j est successeur de x_i si $(x_i, x_j) \in U$; l'ensemble des successeurs de x_i est noté $\Gamma(x_i)$. x_j est prédécesseur de x_i si $(x_j, x_i) \in U$; l'ensemble des prédécesseurs de xi est noté $\Gamma^{-1}(x_i)$.

L'application Γ qui, à tout élément de X, fait correspondre une partie de X est appelée une application multivoque.

1.2 PRINCIPALES DÉFINITIONS

Adjacence : Deux sommets sont adjacents (ou voisins) s'ils sont reliés par un arc. Deux arcs sont adjacents s'ils ont au moins une extrémité commune.

Degrés : Le demi degré extérieur de x_i, $d^+(x_i)$, est le nombre d'arcs ayant x_i comme extrémité initiale. Le demi degré intérieur de x_i, d (x_i), est le nombre d'arcs ayant x_i comme extrémité finale.

Le degré de x_i est $d(x_i) = d^+(x_i) + d^-(x_i)$. Le degré d'un sommet d'un graphe non orienté est le nombre d'arêtes qui lui sont incidentes. Dans le cas d'un 1-graphe, on peut tout aussi bien définir le degré d'un sommet à l'aide de l'application multivoque Γ puisque $d^+(x_i) = |\Gamma(i)|$ et $d^-(x_i) = |\Gamma^{-1}(i)|$.

Graphe complémentaire :

Soient $G=(X, U)$ et $\overline{G} = (X, \overline{U})$ tel que :

$(x_i, x_j) \in U \Rightarrow (x_i, x_j) \notin \overline{U}$ et $(x_i, x_j) \notin U \Rightarrow (x_i, x_j) \notin \overline{U}$. $\overline{G}$ est le graphe complémentaire de G.

Graphe partiel :

$G=(X, U)$ et $Up \subset U$. $Gp=(X, Up)$ est un graphe partiel de G.

Sous graphe :

$G=(X, U)$ et $Xs \subset X$. $Gs= (Xs, V)$ est un sous graphe de G, où V est la restriction de la fonction caractéristique de U à Xs. $\forall x_i \in Xs, \Gamma s(x_i)=\Gamma(x_i) \cap Xs$.

Sous graphe partiel : Combine les deux précédents.

Graphe réflexif : $\forall x_i \in X, (x_i, x_i) \in U$.

Graphe irréflexif : $\forall x_i \in X, (x_i, x_i) \notin U$.

Graphe symétrique : $\forall x_i, x_j \in X, (x_i, x_j) \in U . (x_j, x_i) \in U$.

Graphe asymétrique : $\forall x_i, x_j \in X, (x_i, x_j) \in U . (x_j, x_i) \notin U$ (si G est asymétrique, G est irréflexif).

Graphe antisymétrique : $\forall x_i, x_j \in X, (x_i, x_j) \in U$ et $(x_j, x_i) \in U . x_i = x_j$ (si G est asymétrique, G est aussi antisymétrique).

Graphe transitif : $\forall x_i, x_k, x_j \in X, (x_i, x_j) \in U, (x_j, x_k) \in U . (x_i, x_k) \in U$.

Graphe complet : $\forall x_i, x_j \in X, (x_i, x_j) \notin U . (x_j, x_i) \in U$.

Clique : ensemble des sommets d'un sous graphe complet. Soit C $\subset$ X une clique de G non orienté : $\forall (x_i, x_j) \in C, (x_i, x_j) \in U$ (2 sommets distincts de G sont toujours adjacents). Notons qu'un graphe complet et antisymétrique s'appelle un «tournoi», car il symbolise le résultat d'un tournoi où chaque joueur est opposé une fois à chacun des autres joueurs.

1.3 NOTION DE COMPLEXITÉ D'ALGORITHME

Les problèmes de graphe se rattachent à la grande classe des problèmes d'optimisation combinatoire. Tous ces problèmes se répartissent en deux catégories : ceux qui sont résolus optimalement par des algorithmes linéaires et ceux dont la résolution peut prendre un temps exponentiel sur les grands cas. On parle respectivement d'algorithmes polynomiaux et exponentiels. Il y'a aussi une autre classe médiane d'algorithmes appelée NP-Complet (Not Polynomial). Ce sont des algorithmes dont on a la preuve mathématique qu'ils ne sont pas polynomiaux, mais par contre on ne dispose pas de la preuve pour dire qu'ils sont exponentiels.

Pour évaluer et classer les divers algorithmes disponibles pour un problème de graphe, il nous faut utiliser une mesure de performance indépendante du langage et de l'ordinateur utilisés. Ceci est obtenu par la notion de complexité d'un algorithme, qui consiste à mettre en évidence les possibilités et les limites théoriques du processus calculatoire, en évaluant le nombre d'opérations caractéristiques de l'algorithme dans le pire des cas. Elle est noté O (e.g., $O(n^2)$ pour une fonction qui augmente dans le carré de la taille des données). On rencontre aussi la notation Θ (e.g. $\Theta(n^2)$) qui donne une borne asymptotique par excès et par défaut (alors que O ne donne que la borne asymptotique par excès).

$f(n)=O(g(n)) \Leftrightarrow \exists\ c\in\mathcal{R}, \exists\ N_0\in\mathcal{N} :\ f(n)\geq c\,g(n)\ \forall n>N_0$

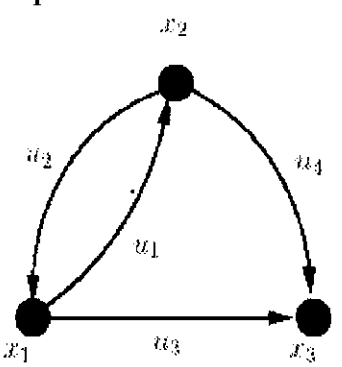
$f(n)=\Theta(g(n)) \Leftrightarrow \exists\ c_1, c_2\in\mathcal{R}, \exists\ N_0\in\mathcal{N} : c_1 g(n)\leq f(n)\leq c_2 g(n)\ \forall n>N_0$

1.4 Représentations d'un graphe

1.4.1 Matrice d'adjacence

On considère un 1-graphe. La matrice d'adjacence fait correspondre les sommets origines des arcs (placés en ligne dans la matrice) aux sommets destinations (placés en colonne). Dans le formalisme matrice booléenne, l'existence d'un arc (x_i, x_j) se traduit par la présence d'un 1 à l'intersection de la ligne x_i et de la colonne x_j ; l'absence par la présence d'un 0 (dans un formalisme matrice aux arcs, les éléments représentent le nom de l'arc).

Exemple :



origine \ destination	x_1	x_2	x_3
x_1	0	1	1
x_2	1	0	1
x_3	0	0	0

Figure 1.6: Représentation par matrice d'adjacence

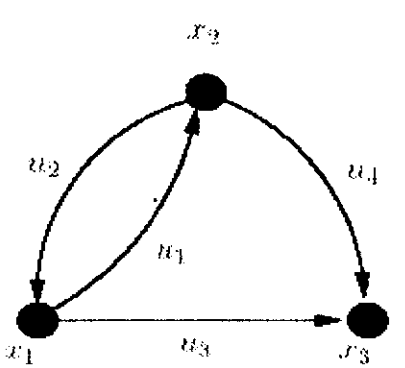
Place mémoire utilisée: n^2 (n nombre de sommet)

1.4.2 Matrice d'incidence sommet–arc

Ligne ↔ sommet, Colonne ↔ arc

Si $u=(i,j)\in U$, on trouve dans la colonne u : $a_{iu} = 1$: $a_{ju} = -1$; tous les autres termes sont nuls.

Exemple :



	u_1	u_2	u_3	u_4
x_1	1	-1	1	0
x_2	-1	1	0	1
x_3	0	0	-1	-1

Figure 1.7: Représentation par matrice d'incidence sommet-arc

Place mémoire utilisée: n x m (m nombre d'arc)

1.4.3 LISTE D'ADJACENCE

Pour un 1–graphe, l'avantage de la représentation par listes d'adjacence (grâce à l'application multivoque Γ), par rapport à celle de la matrice d'adjacence, est le gain obtenu en place mémoire ; ce type de représentation est donc le mieux adapté pour un stockage en mémoire. Le but est de représenter chaque arc par son extrémité finale, son extrémité initiale étant définie implicitement. Tous les arcs émanant d'un même sommet sont liés entre eux dans une liste. A chaque arc sont donc associés le noeud destination et le pointeur au prochain sommet dans la liste. On crée deux tableaux LP (tête de listes) de dimension n+1 et LS (liste de successeurs) de dimension m (cas orienté) ou 2m (cas non orienté). Pour tout i, la liste des successeurs de i est dans le tableau LS à partir de la case numéro LP(i).

1 – On construit LS par Γ(1), Γ(2),..., Γ(n).

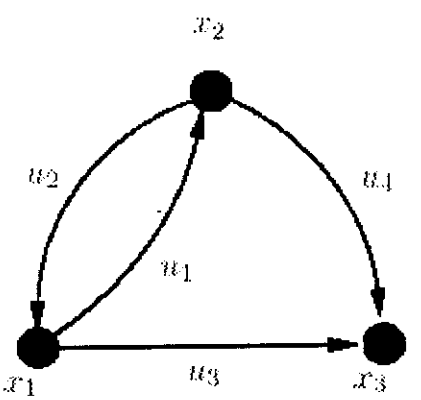
2 – On construit LP qui donne pour tout sommet l'indice dans LS où commencent ses successeurs.

3 – Pour tout sommet i →1er suivant : LS (LP (i)) ; 2ème suivant : LS (LP (i) +1)). L'ensemble des informations relatives au sommet i est contenu entre les cases LP (i) et LP (i+1) –1 du tableau LS.

4 – Si un sommet i n'a pas de successeur, on pose LP (i) = LP (i+1) (liste vide coincée entre les successeurs de i–1 et de i+1).

Pour éviter des tests pour le cas particulier i=n (le sommet i+1 n'existant pas), on « ferme » par convention la dernière liste en posant LP (n+1) = m+1.

Exemple :



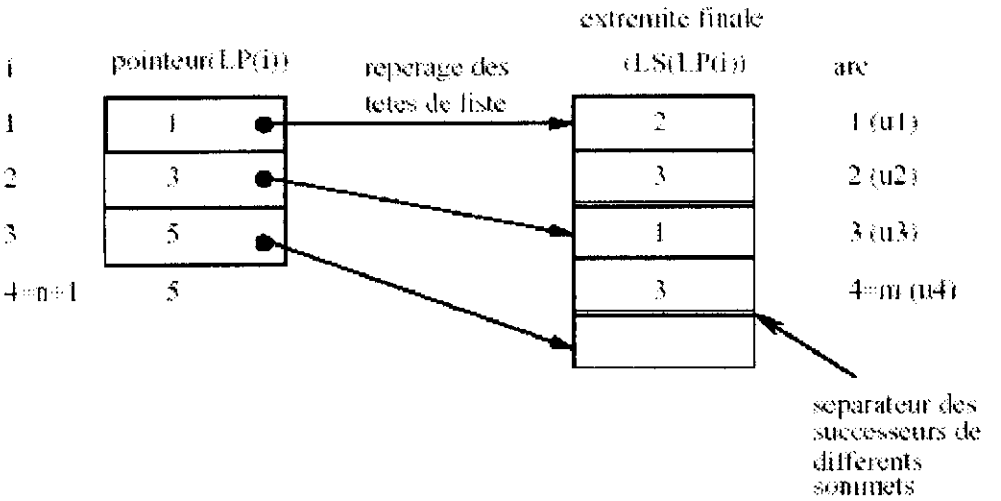


Figure 1.8: Représentation par liste d'adjacence

Place mémoire utilisée n+1+m.

1.5 CONNEXITÉ DANS LES GRAPHES

1.5.1 CHAÎNE – CYCLE

Une chaîne est une séquence d'arc telle que chaque arc ait une extrémité commune avec le suivant.
Un cycle est une chaîne qui contient au moins une arête, telle que toutes les arêtes de la séquences sont différentes et dont les extrémités coïncident.
Exemple :

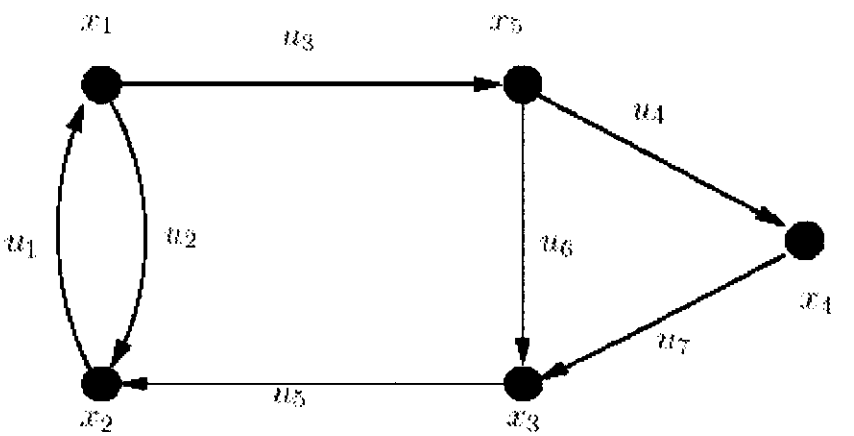


Figure 1.9 :< u2, u5, u6, u4> est une chaîne de x1 à x4.< u4, u7,u6> est un cycle.

1.5.2 CHEMIN – CIRCUIT

Ce sont les mêmes définitions que les précédentes mais en considérant des concepts orientés.

Exemple :(Voir figure 1.8)

$< u_1, u_3, u_4, u_7 >$ est un chemin de x_2 à x_3. $< u_1, u_3, u_6, u_5 >$ est un circuit.

Le sous–ensemble de sommets atteignables à partir d'un sommet donné, grâce à des chemins, est appelé fermeture transitive de ce sommet.

Le terme de parcours regroupe les chemins, les chaînes les circuits et les cycles. Un parcours est :

- *élémentaire* : si tous les sommets qui le composent sont tous distincts ;
- *simple* : si tous les arcs qui le composent sont tous distincts ;
- *hamiltonien* : passe une fois et une seule par chaque sommet du graphe ;
- *eulérien* : passe une fois et une seule par chaque arc du graphe ;
- *préhamiltonien* : passe au moins une fois par chaque sommet du graphe ;
- *préeulérien ou chinois* : passe au moins une fois par chaque arc du graphe.

Le problème du voyageur de commerce est voisin du problème *hamiltonien*. Il consiste à trouver un circuit *hamiltonien* de coût minimal dans un graphe *valué* (pondéré).

1.5.3 CONNEXITÉ

Un graphe G = (X,U) est connexe si $\forall$ i, j $\in$ X, il existe une chaîne entre i et j. On appelle composante connexe le sous-ensemble de sommets tels qu'il existe une chaîne entre deux sommets quelconques.

Un graphe est connexe s'il comporte une composante connexe et une seule. Chaque composante connexe est un graphe connexe.

Exemple :

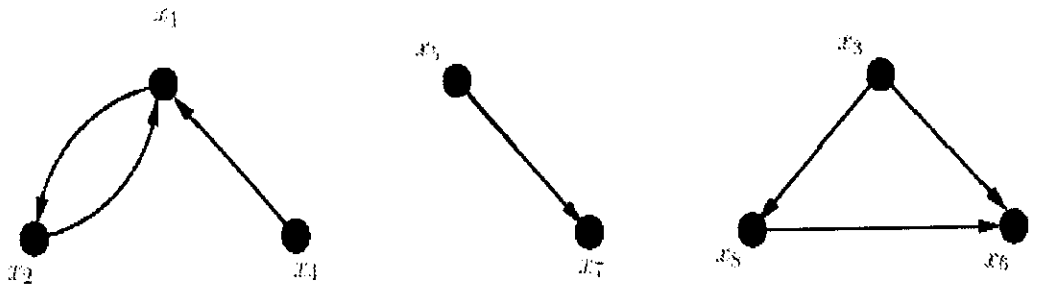


Figure1.10 : Graphe ayant trois composantes connexes.

1.5.4 FORTE CONNEXITÉ

Un graphe G = (X,U) est fortement connexe si $\forall$ i, j $\in$ X, il existe un chemin entre i et j. Une composante fortement connexe (cfc) est un sous-ensemble de sommets tel qu'il existe un chemin entre deux sommets quelconques. Une cfc maximale (cfcm) est un ensemble maximal de cfc. Les différentes cfcm définissent une partition de X.

Un graphe est fortement connexe s'il comporte une seule cfcm.

Recherche de cfcm

Recherche de cfcm

Répéter

1. Partir d'un sommet quelconque n'appartenant pas à une cfcm; le marquer ±.

2. Sur l'ensemble des sommets non marqué ± et tant qu'on peut marquer un sommet faire :

(a) Marquer + tout sommet suivant d'un sommet marqué +.

(b) Marquer – tout sommet précédent d'un sommet marqué –.

Tout sommet marqué ± appartient à une cfcm.

Jusqu' à ce que tout sommet appartienne `a une cfcm.

Complexité : *O(nm).*

Exemple :

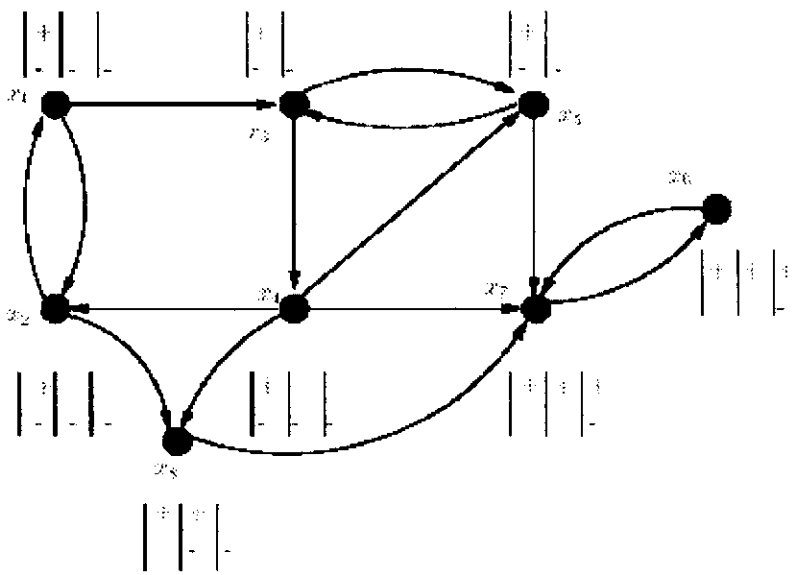


Figure 1.11 :Trois cfcm {x1, x2, x3, x4, x5} , {x6, x7 } , {x8}

On peut trouver une cfcm, à partir de la matrice de fermeture transitive (voir la 2ème partie).

1.6 NOTION DE GRAPHE PONDÉRÉ

On définit un graphe pondéré $G(X, U, \omega)$ comme suit :

– X est l'ensemble des sommets x_i.

– U est l'ensemble des arcs u(i,j) du graphe.

La longueur l_{ij} peut être la distance entre deux villes dans un réseau routier, le coût de transport, les dépenses de construction, le temps de parcours ...

Notons par exemple que pour un réseau routier la distance entre deux villes dépend du chemin pris, le problème du plus court chemin consiste à trouver le chemin qui donne la longueur minimale.

1.7 CONCLUSION

Dans ce chapitre, nous avons introduit quelques notions fondamentales sur la théorie des graphes. Ce qu'il faut retenir pour la suite est la définition d'un graphe, la représentation par matrice d'adjacence, les notions de chemin et de graphes pondérés. Ces notions constitue la base du chapitre suivant qui aborde le coté mathématique du problème de chemin algébrique.

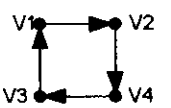
CHAPITRE II

PROBLEME DU CHEMIN ALGÉBRIQUE[1]

Le concept du chemin algébrique formalise et unifie des algorithmes issus de domaines variés, comme la détermination de la fermeture transitive et réflexive d'un graphe, le problème du plus court chemin dans un graphe et l'arbre couvrant de poids minimum. Les algorithmes qui résolvent ces problèmes ont la propriété remarquable d'avoir le même flux de données, ils diffèrent seulement par les opérations élémentaires à effectuer. Nous allons commencer par détailler le problème de fermeture transitive, ensuite nous ferons la généralisation aux autres problèmes.

2.1 FERMETURE TRANSITIVE D'UN GRAPHE

Soit $A = (a_{ij})$ une matrice booléenne qui représente la matrice d'adjacence d'un graphe orienté G à n sommets $\{ x_1, x_2, \ldots, x_n \}$. La fermeture transitive et réflexive de A est la matrice n×n $A^* = (a_{ij}^*)$, où a_{ij}^* est égale à 1 si et seulement si il existe dans le graphe un chemin de longueur non nulle du sommet x_i au sommet x_j. Si A représente G, alors A+I, où I est la matrice identité représente les chemin de longueur 0 ou 1. En d'autre termes A+I a un 1 en position (i, j) si et seulement si i = j ou s'il existe un arc de x_i à x_j. De même, $(A+I)^2$ représente les chemins de longueur 2 ou moins, $(A+I)^3$ représente les chemin de longueur 3 ou moins, et ainsi de suite.



$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Matrice A + I *Matrice $(A+I)^2$* *Matrice $(A+I)^4$*

Figure 2.1 : Fermeture transitive d'un graphe 4×4

[1] Ce chapitre a été rédigé en s'appuyant sur la référence[3]

Finalement $A^* = (A+I)^n$, la matrice A^* peut être obtenue en calculant $(A+I)^2$, $(A+I)^4$, $(A+I)^8$, ... au lieu de calculer $(A+I)^2$, $(A+I)^3$, $(A+I)^4$, $(A+I)^5$, ... pour converger plus rapidement vers A^*. Le calcul se fait donc en $k = \log_2(n)$ multiplications successives de matrices booléennes.

Cependant il existe un algorithme adapté aux architectures systoliques qui se fait en n (n taille de la matrice A) phases.

La première phase consiste à :

$a_{ij}^{(1)} = a_{ij} + a_{i1} * a_{1j}$ i, j ≠ 1

où + désigne l'opération *OU* booléenne, * l'opération *ET* booléenne.

La deuxième phase consiste à :

$a_{ij}^{(2)} = a_{ij}^{(1)} + a_{i2}^{(1)} * a_{2j}^{(1)}$ ∀i, j ≠ 2

La troisième phase consiste à :

$a_{ij}^{(3)} = a_{ij}^{(2)} + a_{i3}^{(2)} * a_{3j}^{(3)}$ ∀i, j ≠ 3

.....

La $k^{ième}$ phase consiste à :

$a_{ij}^{(k)} = a_{ij}^{(k-1)} + a_{ik}^{(k-1)} * a_{kj}^{(k-1)}$ $\forall i, j \neq k$

.....

La $n^{ième}$ phase consiste à :

$a_{ij}^{(n)} = a_{ij}^{(n-1)} + a_{in}^{(n-1)} * a_{nj}^{(n-1)}$ ∀i, j ≠ n

On peut démontrer par récurrence que cet algorithme calcule la matrice de fermeture transitive. A chaque phase $a_{kk}^{(k)} = a_{kk}^{(k-1)}$

$a_{ij}^{(0)} = 1$ si et seulement s'il existe un chemin directe de i à j.

$a_{ij}^{(1)} = 1$ si et seulement s'il existe un chemin directe de i à j, ou passant par le sommet 1.

$a_{ij}^{(2)} = a_{ij} + a_{i1} * a_{1j} + \{a_{i2} + a_{i1} * a_{12}\} * \{a_{2j} + a_{21} * a_{1j}\}$

$a_{ij}^{(2)} = a_{ij} + a_{i1} * a_{1j} + a_{i2} * a_{2j} + a_{i2} * a_{21} * a_{1j} + a_{i1} * a_{12} * a_{2j} + a_{i1} * a_{12} * a_{21} * a_{1j}$

On remarque que $a_{ij}^{(2)} = 1$, s'il existe un chemin de i à j passant par les sommet {1,2}

On suppose que l'hypothèse de récurrence est vérifiée jusqu'à l'ordre k–1,

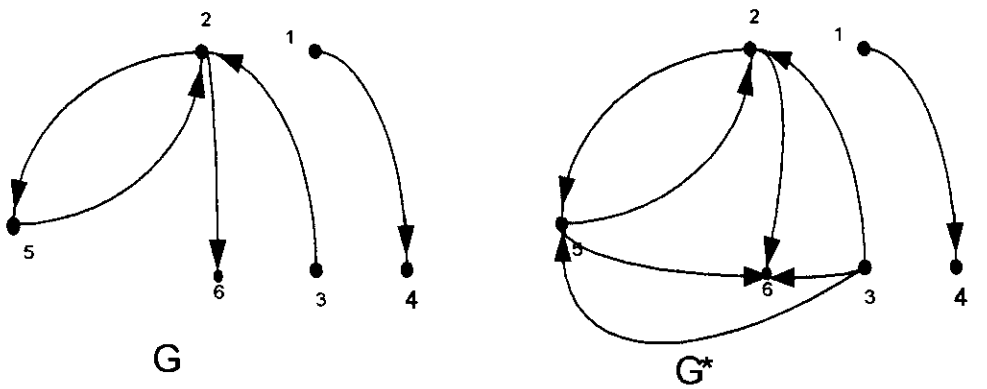
$a_{ij}^{(k)} = a_{ij}^{(k-1)} + a_{ik}^{(k-1)} * a_{kj}^{(k-1)}$ $\forall i, j \neq k$

$a_{ij}^{(k)} = 1$, si et seulement si $a_{ij}^{(k-1)} = 1$ (i.e. il existe un chemin de i à j et passant par {1,2,...,k–1}), ou $a_{ik}^{(k-1)} = 1$ et $a_{kj}^{(k-1)} = 1$ i.e. il existe un chemin de i à k passant par

{1,2,...,k–1} et un chemin de k à j passant par {1,2,...,k–1} , en d'autre termes s'il existe un chemin de i à j passant par {1,2,...,k–1,k}

En fin de compte $a_{ij}^{(n)} = 1$, s'il existe un chemin allant de i à j et passant par {1,2,...,n} (tous les chemins envisageables)

Exemple :



$$\begin{bmatrix}1&0&0&1&0&0\\0&1&0&0&1&1\\0&1&1&0&0&0\\0&0&0&1&0&0\\0&1&0&0&1&0\\0&0&0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&1&0&0\\0&1&0&0&1&1\\0&1&1&0&0&0\\0&0&0&1&0&0\\0&1&0&0&1&0\\0&0&0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&1&0&0\\0&1&0&0&1&1\\0&1&1&0&1&1\\0&0&0&1&0&0\\0&1&0&0&1&1\\0&0&0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&1&0&0\\0&1&0&0&1&1\\0&1&1&0&1&1\\0&0&0&1&0&0\\0&1&0&0&1&1\\0&0&0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&1&0&0\\0&1&0&0&1&1\\0&1&1&0&1&1\\0&0&0&1&0&0\\0&1&0&0&1&1\\0&0&0&0&0&1\end{bmatrix}$$

A $\quad$ *phase* 1 $\quad$ *phase*2 $\quad$ *phase*3 $\quad$ *phase*4

$$\begin{bmatrix}1&0&0&1&0&0\\0&1&0&0&1&1\\0&1&1&0&1&1\\0&0&0&1&0&0\\0&1&0&0&1&1\\0&0&0&0&0&1\end{bmatrix}\qquad\begin{bmatrix}1&0&0&1&0&0\\0&1&0&0&1&1\\0&1&1&0&0&0\\0&0&0&1&0&0\\0&1&0&0&1&0\\0&0&0&0&0&1\end{bmatrix}$$

*phase*5 $\quad$ *phase*6 = A^*

Figure 2.2 : Fermeture transitive d'un graphe 6×6

2.2 LONGUEUR DU CHEMIN LE PLUS COURT DANS UN GRAPHE

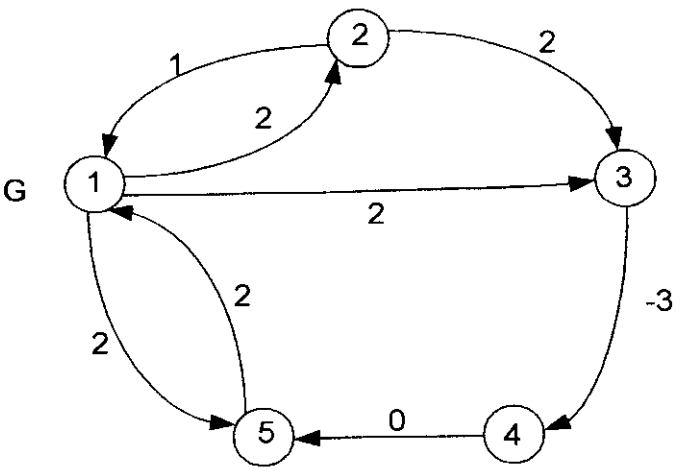
Soit un graphe pondéré $G(X,U,\omega)$, représenté par la matrice $W = (w_{ij})$ telle que $w_{ij} = l_{ij}$ (longueur du chemin directe de i à j, $w_{ii} = 0$), $w_{ij} = -\infty$ s'il n'y a pas de chemin directe allant de i à j.

Il s'agit de trouver pour chaque couple (i,j) la longueur du chemin le plus court allant de i à j. Pour cela on utilisera l'algorithme précédant, en remplaçant l'opération *OU* par l'opération *MIN*, et l'opération *ET* par l'addition. On obtient donc l'équation de récurrence suivante :

$$w_{ij}^{(k)} = \min(w_{ij}^{(k-1)}, w_{ik}^{(k-1)} + w_{kj}^{(k-1)})$$

A chaque phase on calculera la longueur du chemin minimum passant par les sommets {1,2,...,k}.

Exemple :



$$W^{(0)} = \begin{bmatrix} 0 & 2 & 2 & \infty & 2 \\ 1 & 0 & 2 & \infty & \infty \\ \infty & \infty & 0 & -3 & \infty \\ \infty & \infty & \infty & 0 & 0 \\ 1 & \infty & \infty & \infty & 0 \end{bmatrix} \quad W^{(1)} = \begin{bmatrix} 0 & 2 & 2 & \infty & 2 \\ 1 & 0 & 2 & \infty & 3 \\ \infty & \infty & 0 & -3 & \infty \\ \infty & \infty & \infty & 0 & 0 \\ 1 & 3 & 3 & \infty & 0 \end{bmatrix} \quad W^{(2)} = \begin{bmatrix} 0 & 2 & 2 & \infty & 2 \\ 1 & 0 & 2 & \infty & 3 \\ \infty & \infty & 0 & -3 & \infty \\ \infty & \infty & \infty & 0 & 0 \\ 1 & 3 & 3 & \infty & 0 \end{bmatrix}$$

$$W^{(3)} = \begin{bmatrix} 0 & 2 & 2 & -1 & 2 \\ 1 & 0 & 2 & -1 & 3 \\ \infty & \infty & 0 & -3 & \infty \\ \infty & \infty & \infty & 0 & 0 \\ 1 & 3 & 3 & 0 & 0 \end{bmatrix} \quad W^{(4)} = \begin{bmatrix} 0 & 2 & 2 & -1 & -1 \\ 1 & 0 & 2 & -1 & -1 \\ \infty & \infty & 0 & -3 & -3 \\ \infty & \infty & \infty & 0 & 0 \\ 1 & 3 & 3 & 0 & 0 \end{bmatrix} \quad W^{(5)} = \begin{bmatrix} 0 & 2 & 2 & -1 & -1 \\ 0 & 0 & 2 & -1 & -1 \\ -2 & 0 & 0 & -3 & -3 \\ 1 & 3 & 3 & 0 & 0 \\ 1 & 3 & 3 & 0 & 0 \end{bmatrix}$$

Figure 2.3: Exemple du calcul de la longueur du chemin le plus court pour un graphe 6×6

2.3 ARBRE COUVRANT DE POIDS MINIMUM

Un arbre est un graphe non orienté, connexe, acyclique

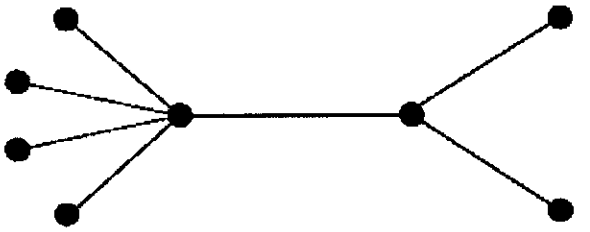


Figure 2.4 : Exemple d'un arbre

On dit qu'un arbre couvre le graphe G s'il connecte tous ses sommets. Un arbre couvrant de poids minimum est celui dont le coût (total) de connexion est minimal. L'application principale de cet algorithme réside dans l'optimisation des : réseaux informatique, câblage de circuits électroniques, lignes à hautes tension, oléoduc...

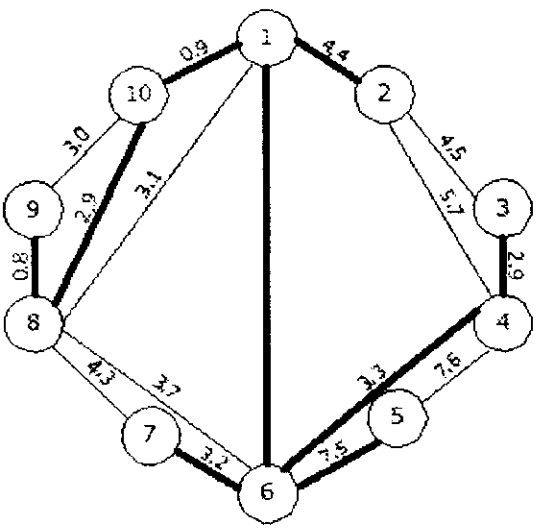


Figure 2.5 : Graphe à 10 nœuds et son arbre couvrant de poids minimum

Le graphe est représenté par une matrice W telle w_{ij} représente le poids de l'arrête (coût de la connexion entre i et j), qui peut être infini si la connexion (i,j) n'est pas envisageable.

Avant d'exposer l'algorithme on va démontrer un lemme qui servira pour la suite.

LEMME

Si tous les w_{ij} sont inégales, alors l'arc (i,j) fait partie de l'arbre couvrant de poids minimum de G, si et seulement si tout chemin menant de i à j et contenant 2 arrêtes ou plus, contient une arrête de poids supérieur à w_{ij}.

PREUVE

Soit w_{ij} le poids de l'arc (i,j) pour $1 \leq i,j \leq N$, et soit T l'arbre couvrant de poids minimum de G. On commence par démontrer que si $(i,j) \in T$, alors chaque chemin contenant deux arrêtes ou plus de i à j contient une arrête de poids supérieur à w_{ij}. Cette preuve se fait par l'absurde, i.e. on suppose que $(i,j) \in T$, et qu'il existe un chemin P_{ij} menant de i à j telle que toutes ses arrêtes sont de poids inférieure à w_{ij}.

Donc si on supprime l'arête (i,j) de T on obtient deux arbres disjoints T_i et T_j contenant respectivement les nœuds i et j, et qui collectivement couvrent tout le graphe. Du moment que P_{ij} lie i et j, il doit contenir une arrête (i',j') qui lie T_i et T_j. Si on définit $T'=T_i \cup T_j \cup (i',j')$, on trouve que c'est un arbre couvrant – il contient N-1 arrête et est acyclique, de plus il est à coût minimum, ce qui contredit l'hypothèse de départ T arbre couvrant de poids minimum. Pour l'autre direction (de l'équivalence), on suppose que tout chemin de deux arrêtes ou plus de i à j contient une arrête de poids supérieur à w_{ij}, mais que $(i,j) \notin T$. Soit (i',j') une arrête de poids supérieure à w_{ij}, et soit $T_{i'}$ et $T_{j'}$ les arbres partiels de T obtenus en supprimant l'arrête (i',j') de T. L'un des deux arbres contiendra i et l'autre j, de même si on définit $T'=T_{i'} \cup T_{j'} \cup (i,j)$, on trouve que c'est un arbre couvrant de poids minimum, ce qui contredit l'hypothèse de départ.

Après la démonstration de ce lemme il est aisé de tirer un algorithme pour trouver l'arbre couvrant de poids minimum. En réalité il sera identique à celui du chemin le plus court, excepté le fait que le poids d'un chemin est la valeur du poids de l'arrête la plus « lourde », au lieu de la somme des poids.

$w_{ij}^{(k)} = \min(w_{ij}^{(k-1)}, \max(w_{ik}^{(k-1)}, w_{kj}^{(k-1)}))$ $k=1...n$, $1 \le i,j \le N$.

(i,j) fera alors partie de l'arbre couvrant de poids minimum si et seulement si $w_{ij} = w_{ij}^{(n)}$.

2.4 CONCLUSION

Nous avons abordé dans ce chapitre le coté mathématique du problème de chemin algébrique en démontrant l'algorithme de Warshall-Floyd pour la fermeture transitive puis en généralisant cet algorithme pour la longueur du plus court chemin entre deux sommets d'un graphe ainsi que pour l'arbre couvrant de poids minimum. Avant d'aborder la traduction de cet algorithme en circuit, quelques notions sur les architectures systoliques s'imposent.

CHAPITRE III

ARCHITECTURES SYSTOLIQUES[1]

Les réseaux systoliques sont des processeurs intégrés spécialisés dont les caractéristiques dominantes sont :

- un parallélisme massif,
- des communications locales,
- un mode opératoire synchrone.

Nous commençons par préciser cette définition, en expliquant ce qui a motivé le développement de telles architectures.

L'évolution de la technologie des circuits intégrés est une notion fondamentale lorsqu'on s'intéresse à l'architecture des machines. Pendant longtemps, l'augmentation de la vitesse des circuits était telle qu'en suivant l'évolution des composants, il suffisait aux constructeurs de conserver l'architectures séquentielle des machines (figure 3.1) pour pouvoir mettre sur le marché des ordinateurs de plus en plus puissants.

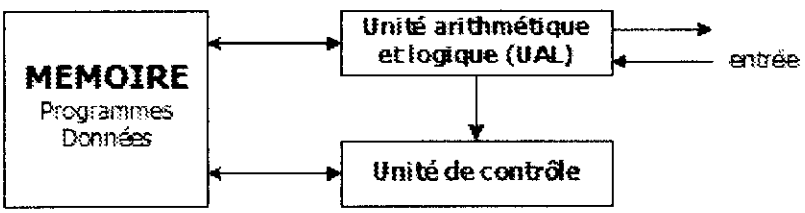


Figure 3.1: Modèle séquentiel

3.1 LE PIPELINE

La technique de pipeline a été appliquée pour la première fois dans l'industrie automobile par Henri Ford au début du 20ième siècle.

[1] Ce chapitre a été rédigé en s'appuyant sur la référence [4]

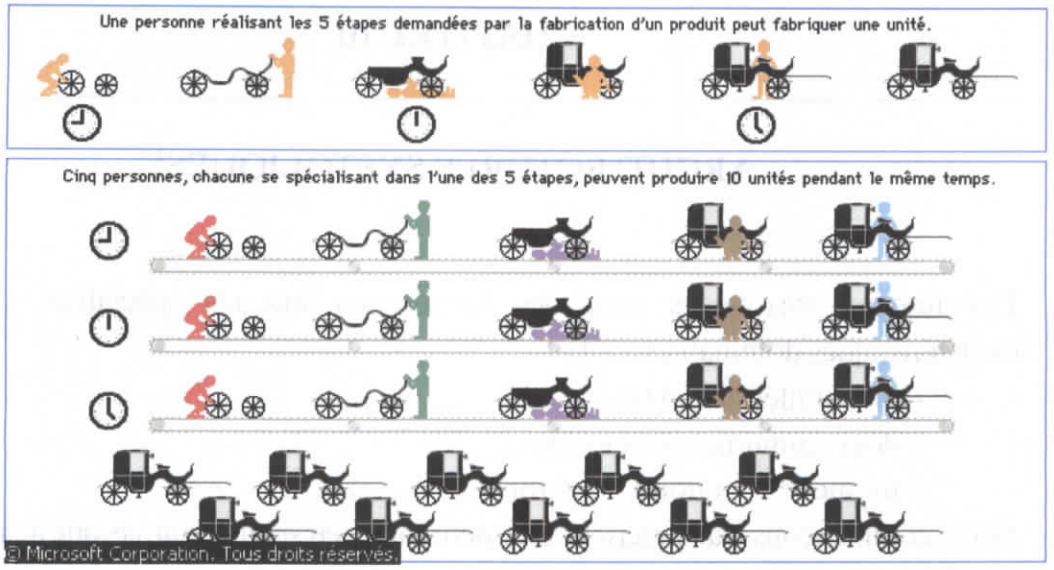


Figure 3.2 : Notion de pipeline

Elle a été ensuite utilisée dans d'autres domaines, dont l'informatique.
Si on admet qu'une instruction est exécutée en 3 cycles qui sont :

- Recherche de l'instruction dans la mémoire (FETCH)
- Décodage de l'instruction
- Exécution de l'instruction

On obtient la structure suivante :

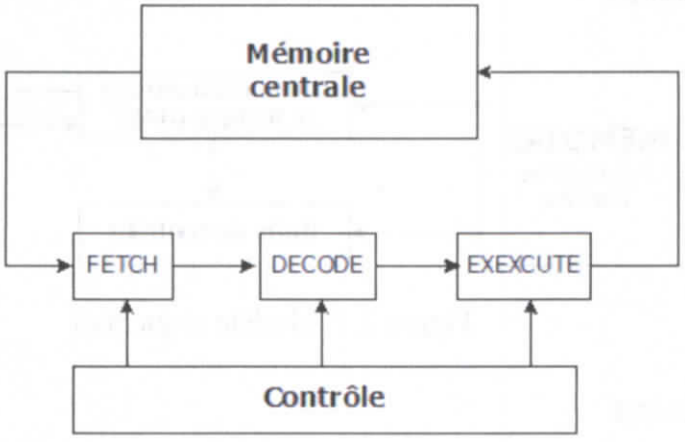


Figure 3.3 : Fonctionnement du modèle vectoriel

A chaque instant t, le bloc FETCH recherche l'instruction i, le bloc DECODE décode l'instruction i–1, et le bloc EXECUTE se charge de l'exécution de l'instruction i–2 .

L'exploitation des structures pipelines apporte un gain important en performances, sans pour autant nécessiter des modifications profondes de la structure des programmes. Les techniques de vectorisation permettent de tirer parti de l'organisation en pipeline du calcul des opérations arithmétiques élémentaires. C'est la manifestation la plus marquante d'un syndrome plus général : peu de parties de

l'ordinateur ont résisté à la « pipelinisation ». Conçus pour effectuer de nombreuses opérations identiques sur de longues séquences de données, les calculateurs vectoriels n'ont cessé d'accroître leur puissance durant les deux dernières décennies. Cependant l'augmentation des performances, pour importantes qu'elle soit, n'est plus suffisante pour répondre aux besoins des utilisateurs. A cela plusieurs raisons : tout d'abord, l'industrie des semi conducteurs est, d'une manière générale, plus à même de développer des technologies à très haute densité d'intégration que de réaliser des circuits ultra rapides. Ensuite on a atteint les limites physiques qui semblent incontournables jusqu'à l'arrivée sur le marché de technologies très rapides (à l'Arséniure de Galium *AsGa*); enfin le prix à payer pour gagner un ordre de grandeur sur la vitesse des composants en utilisant ces nouvelles technologies est prohibitif pour la plus part des applications.
La réponse apportée par tous les constructeurs de superordinateurs à la demande accrue des utilisateurs tient en un mot : le parallélisme.

3.2 LE PARALLÉLISME

Le concept de parallélisme rompt avec l'approche classique qui consiste à gagner de la vitesse en effectuant plus rapidement chaque opération. En calcul parallèle, le gain en vitesse provient de la réalisation simultanée de plusieurs opérations. Il ne faut pas croire que le recours au parallélisme pour accélérer des calculs soit une idée nouvelle, mais ce n'est que récemment que la technologie a permis de construire des multiprocesseurs et de les utiliser efficacement.
Comme leur nom l'indique, les multiprocesseurs sont des ordinateurs possédants plusieurs processeurs (2 à plusieurs dizaines pour des systèmes généraux ou beaucoup plus pour des machines spécialisées). L'architecture se complique : ainsi les problèmes d'accès à la mémoire deviennent cruciaux pour pouvoir acheminer des données au rythme de traitement des processeurs; de même les problèmes de communication et de synchronisation entre processeurs sont importants.
Plusieurs modèles d'architectures parallèles ont été proposés. Pour les classifier, un bon critère de sélection : le modèle de contrôle des suites d'opérations élémentaires effectuées par les différents processeurs. On obtient les architectures suivantes, selon la multiplicité des flux d'instructions et de données disponibles matériellement (classification de Flynn) :

SIMD : Single Instruction Multiple Data
Un seul flux d'instruction, plusieurs flux de données.
MIMD : Multiple Instruction Multiple Data
Plusieurs flux d'instruction, plusieurs flux de données

MISD : Multiple Instruction Single Data

Plusieurs flux d'instruction, une seule instruction

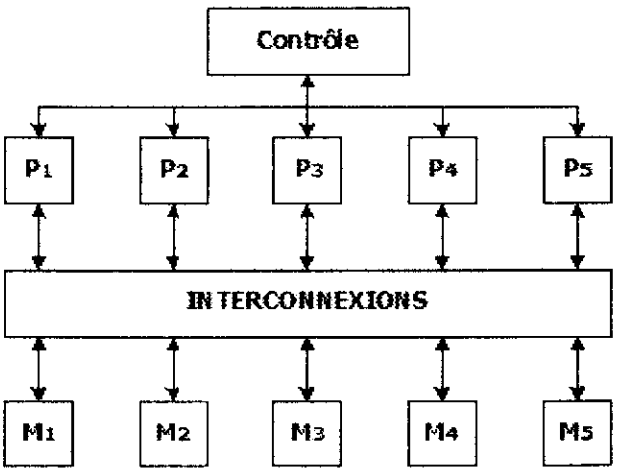


Figure 3.4 : Architecture SIMD

Dans le modèle SIMD, plusieurs unités de traitement sont supervisées par une même unité de contrôle. Toutes les unités de traitement reçoivent la même instruction (ou le même programme) diffusée par l'unité de contrôle, mais opèrent sur des ensembles de données distincts, provenant de flux de données distincts.
Chaque processeur exécutant la même instruction au même instant, on obtient un fonctionnement synchrone. La mémoire partagée peut être subdivisée en plusieurs modules. Dans ce cas, l'accès des unités de traitement aux différents modules se fait par un réseau d'interconnexion.

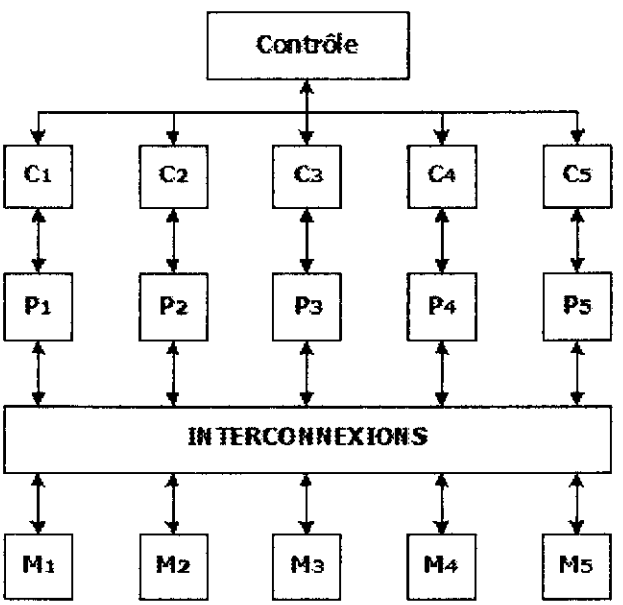


Figure 3.5 : Architecture MIMD

La différence profonde entre le modèle MIMD et le précèdent est le fait que, dans ce cas chaque processeur possède sa propre unité de contrôle. Les processeurs ont donc un fonctionnement indépendant (en particulier asynchrone) et exécutent des programmes différents. Il reste une certaine centralisation dans le fonctionnement des machines MIMD à faible nombre de processeurs. Celles-ci fonctionnent par partage de données. Les unités de traitement accèdent à une mémoire partagée par l'intermédiaire d'interconnexion et communiquent en échangeant des données (exemple : L'ALLIANTE FX/8).

Mais pour une architecture comportant un très grand nombre de processeurs (ici encore, poussés par la demande des utilisateurs, des constructeurs évoluent vers des machines comportant des processeurs toujours plus importants), il est impératif d'évoluer vers un fonctionnement totalement décentralisé. Dans le fonctionnement par échange de messages, chacune des unités de traitement possède sa mémoire propre (mémoire locale) et communique avec ses voisins en échangent des messages (exemple : l'hypercube iPSC d'Intel).

Quand au modèle MISD, une même donnée est distribuée vers différents processeurs pour subir un traitement parallèle (voir le chapitre 5).

3.3 LE PRINCIPE DE LOCALITÉ

Lorsqu'on cherche à tirer partie d'une architecture parallèle pour exécuter un algorithme donné, le problème principal peut se résumer de la façon suivante : comment répartir l'algorithme sur les processeurs, de telle sorte qu'ils soient tous actifs en permanence pendant la durée de l'algorithme, et ce, en effectuant un travail utile ?

En examinant de près ce problème dans la réalité, on s'aperçoit rapidement que la condition principale de succès n'est pas de disposer de processeurs qui calculent vite, mais plutôt que ces processeurs puissent communiquer entre eux de façon efficace, c'est à dire qu'ils puissent échanger de l'information aussi rapidement qu'ils accèdent (et opèrent sur) à leurs propres données.

Ici intervient une donnée essentielle de la technologie actuelle. Il est impossible de réaliser une machine ayant un nombre de processeurs élevés (disons 100) où tous les processeurs peuvent communiquer deux à deux de façon directe (réseau à maillage complet).

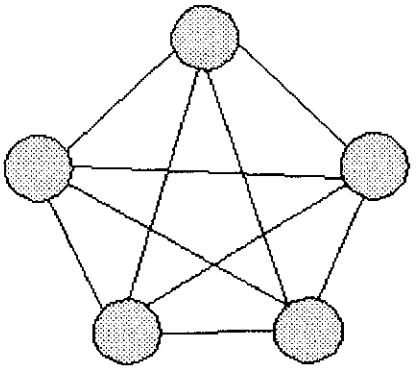


Figure 3.6 : Réseau de processeurs totalement connectés

Tout réseau de communication fait intervenir d'une façon ou d'une autre le concept de localité : pour une répartition spatiale donnée de processeurs, il n'est pas possible de relier efficacement un processeur qu'avec un voisinage de celui-ci. En conséquence, un algorithme ne pourra être programmé de façon efficace sur la machine que si son exécution peut être répartie de telle sorte que chaque processeur n'ait à communiquer qu'avec ses voisins, au sens du réseau de communication de l'architecture. Il s'agit là d'une propriété très forte.

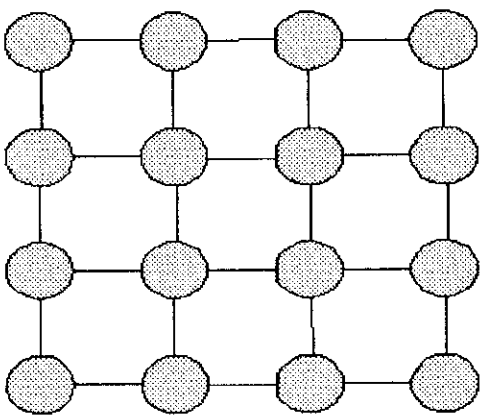


Figure 3.7 : Réseau de processeurs connectés localement

Une seconde conséquence du mode de connexion local concerne le rapport entre entrée-sortie et calcul. Puisqu'il n'est pas possible de relier chaque processeur à tout les autres, il est de la même façon difficile d'imaginer que chaque processeur soit relié directement à l'environnement extérieur de la machine, qu'on appelle l'hôte du système, censé fournir les données de l'algorithme, et centraliser ces résultats. D'une façon ou d'une autre, cela signifie qu'il faut que la complexité de l'algorithme repose sur les calculs, et non sur la quantité d'entrées-sorties qu'il doit effectuer.

Pour mieux faire comprendre ceci, imaginons un algorithme qui s'exécute avec n processeurs, nécessite n opérations et n entrées-sorties, comme par exemple le produit d'un vecteur de n composantes par une constante. Pour que l'algorithme

puisse être exécuté efficacement, il faut que tous les processeurs puissent lire simultanément une donnée et fournir simultanément un résultat, sinon le temps dominant serait toujours celui nécessaire aux entrées-sorties.

Cette caractéristique fondamentale de la technologie actuelle se retrouve évidemment lorsqu'on s'intéresse aux circuits intégrés. Ceux ci sont par nature planaires, ou mieux tridimensionnel, et chaque élément de calcul ne peut communiquer efficacement qu'avec ses voisins du plan ou de l'espace. De plus il n'est possible de communiquer avec un circuit que par l'intermédiaire de ses éléments externes, ce qui dans le meilleur des cas permet d'effectuer $\sqrt{n}$ entrées-sorties simultanément lorsque n processeurs sont disposés dans le plan.

3.4 LE MODÈLE SYSTOLIQUE

La complexité des circuits intégrés disponible à l'heure actuelle rend possible la réalisation à faible coût de systèmes massivement parallèles, pour peu que le mode de communication des processeurs respecte la contrainte de localité qu'on vient d'énoncer. De même la classe d'application est bien délimitée : comme on l'a dit, le volume de calculs à effectuer doit primer largement sur les transferts de données à réaliser. Par suite plutôt que les calculateurs généraux, il s'agit de processeurs spécialisés que l'on adjoint à un processeur hôte de type conventionnel.

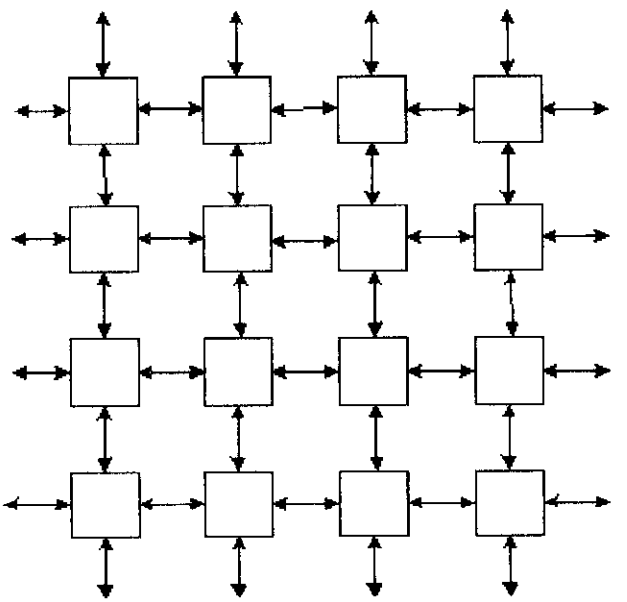


Figure 3.8 : Réseau orthogonal

Le modèle systolique introduit en 1978 par Kung et Leiserson, s'est révélé être un outil puissant pour la conception de processeurs intégrés spécialisés. En un mot une architecture systolique est agencée en forme de réseau. Ces réseaux se composent d'un grand nombre de cellules élémentaires identiques et localement

interconnectées. Chaque cellule reçoit des données des cellules voisines, effectue un calcul simple, puis transmet les résultas toujours aux cellules voisines, un temps de cycle plus tard. Seul les cellules situées à la frontière communiquent avec l'environnement extérieur. Pour fixer les idées, disons que chaque cellule a la complexité au plus d'un petit microprocesseur.

Les cellules évoluent en parallèle, sous le contrôle d'une horloge globale (synchronisme total) : plusieurs calculs sont effectués simultanément sur le réseau, et on peut pipeliner la résolution de plusieurs instances du problème sur le réseau. La dénomination « systolique » provient d'une analogie entre la circulation des flots de données dans le réseau et de celle du sang humain, l'horloge qui assure la synchronisation globale du système constituant le « cœur » du système. En plus de la propriété de localité, une propriété remarquable des architectures systoliques est la régularité, il suffit de connaître la topologie d'une portion de l'architecture (quelques EP) pour déduire la structure de l'architecture globale, car celle-ci n'est que la duplication dans les deux dimensions (voir trois) de la portion élémentaire.

" Systolic systems are an attempt to capture the concepts of parallelism, pipelining and interconnection structure in a unified framework of mathematics and engineering. They embody engineering techniques such as multiprocessing and pipelining together with more theoretical ideas of cellular automata and algorithms, and therefore are an excellent subject of investigation form a combined standpoint"

C.E.Leiserson, PhD Thesis, Carneige Mellon University, USA 1981

3.5 NOTIONS ÉLÉMENTAIRES

Latency: Elle correspond au temps nécessaire pour que la donnée traverse, le chemin critique de l'architecture, elle peut être décomposée en L_{de}, L_{dd}, L_{dp}

Où:

- L_{de} (data entry time) : C'est le temps qui s'écoule entre l'entrée du premier élément et celle du dernier sur le chemin critique.
- L_{dd}(data delay time) : C'est le nombre d'étape qui précède l'entrée du premier élément sur le chemin critique.
- L_{dp} (data processing time) : C'est le temps requis par chaque donnée pour être traité par chaque EP du chemin critique.

I/O bandwidth: C'est le nombre d'éléments qui peuvent être échangés entre l'architecture systolique et le système hôte à chaque cycle d'horloge.

Scalability: C'est la capacité de maintenir la latency linéairement proportionnelle au nombre d'EP (Elementary Processor) quelque soit la dimension du problème.

Problem-size-independance (ψ): Une architecture parallèle est dite ψ, si et seulement si, disposant d'une architecture de taille finie, n'importe quelle taille du problème peut être résolu à l'aide de cette architecture.
Ceci implique que le traitement peut être décomposé, chaque partie résolue à l'aide de l'architecture matérielle disponible, puis les résultats doivent être assemblés pour donner le résultat global.
Cette notion reflète une des contraintes à laquelle on doit faire face dans la pratique où souvent, la dimension du problème dépasse la dimension du matérielle disponible.
Efficacité: C'est une mesure qui peut prendre une valeur comprise entre 0 et1.
Il existe plusieurs formules qui donnent cette mesure, elles expriment toutes la même chose mais différemment. La plus utilisée est E=S/P où S (Speedup) est le gain en vitesse obtenu par rapport à l'algorithme séquentielle, et P est le nombre d'EP utilisés. Soit par exemple un algorithme séquentielle qui effectue la multiplication de deux matrices N×N en N^3 cycles d'horloge, et soit une architecture parallèle capable de faire cette multiplication en un N cycles d'horloges en utilisant N^2 EP alors $S=N^3/N=N^2$, $E=N^2/N^2=1$. Ceci signifie qu'on ne peut faire mieux en matière de parallélisme.

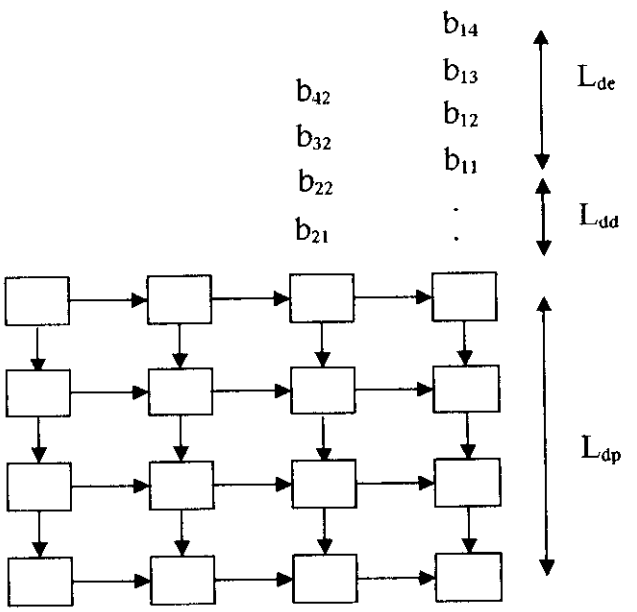


Figure 3.9 :Exemple de la latency d'une architecture systolique

3.6 TYPES D'ARCHITECTURES SYSTOLIQUES

Systolique pure: C'est une architecture parallèle qui vérifie les contraintes de régularité et de localité.
Full systolique: C'est une architecture parallèle qui vérifie :

- Les entrées /sorties sont assurées par les cellules situées à la périphérie du réseau.
- Les cellules élémentaires (EP) ne nécessitent pas un préchargement.

Semi systolique : C'est une architecture parallèle qui vérifie :

- Les cellules élémentaires nécessitent un préchargement avant le lancement des opérations de calcul.

3.7 CONCLUSION

Dans ce chapitre, nous avons présenté les architectures systoliques et leurs principales propriétés. On retient essentiellement un parallélisme massif, un flot de données important et un synchronisme total.

La cible technologique qui supportera de telles architectures est décrite dans le prochain chapitre.

CHAPITRE IV

CIRCUITS PROGRAMMABLES

L'électronique moderne se tourne de plus en plus vers le numérique qui présente de nombreux avantages sur l'analogique : grande insensibilité aux parasites et aux dérives diverses, (re)configurabilité, facilité de stockage de l'information etc....

D'abord réalisées avec des circuits SSI (Small Scale Integration) les fonctions logiques intégrées se sont développées avec la mise au point du transistor MOS dont la facilité d'intégration a permis la réalisation de circuits MSI (Medium Scale Integration) puis LSI (Large Scale Integration) puis VLSI (Very Large Scale Integration). Ces deux dernières générations ont vu l'avènement des microprocesseurs et microcontrôleurs.

Au début des années 70 sont apparus les premiers composants (en technologie bipolaire) entièrement configurables par programmation. La nouveauté résidait dans le fait qu'il était maintenant possible d'implanter physiquement par simple programmation, au sein du circuit, n'importe quelle fonction logique, et non plus de se contenter de faire réaliser une opération logique par un microprocesseur dont l'architecture est figée.

D'abord dédiés à des fonctions simples en combinatoire (décodage d'adresses par exemple), ces circuits laissent aujourd'hui au concepteur la possibilité d'implanter des composants aussi divers qu'un inverseur et un microprocesseur au sein d'un même boîtier . L'intégration des principales fonctions d'une carte dans un même boîtier permet de répondre à la fois aux critères de densité et de rapidité (les capacités parasites étant plus faibles, la vitesse de fonctionnement peut augmenter).

La plupart de ces circuits sont maintenant programmés à partir d'un simple ordinateur type PC directement sur la carte où ils vont être utilisés. En cas d'erreur, ils sont programmables électriquement sans avoir à extraire le composant de son environnement.

Le terme même de circuit programmable est ambigu, la programmation d'un FPGA ne faisant pas appel aux mêmes opérations que celle d'un microprocesseur. Il serait plus juste de parler pour les PLD, CPLD et FPGA de circuits à architecture programmable ou encore de circuits à réseaux logiques programmables.

Parallèlement à ces circuits, on trouvera les ASIC (Application Specific Integrated Circuits) qui sont des composants où le concepteur intervient au niveau du dessin de la pastille de silicium en fournissant des masques à un fondeur. On ne peut plus

franchement parler de circuits programmables. Les temps de développement long ne justifient l'utilisation que pour des grandes séries.
Donc nous pouvons classer les circuits numériques comme suit [5] :

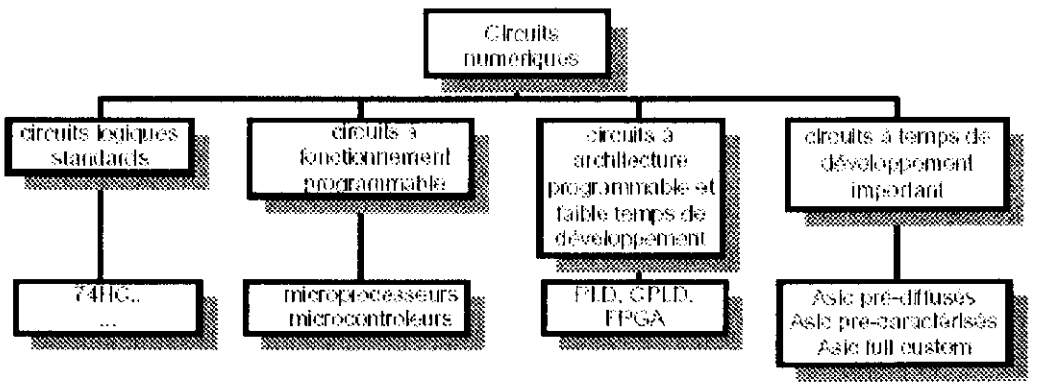


Figure 4.1 : Classifications des circuits numériques

Nous nous intéresserons aux circuits à architecture programmable à faible temps de développement.

4.1 LES CIRCUITS A ARCHITECTURE PROGRAMMABLE

Ces circuits ne nécessitent aucune étape technologique supplémentaire pour être personnalisés. Ce sont des circuits standards programmables par l'utilisateur grâce à différents outils de développement. Nous y trouvons les :

- PAL (Programmable Array Logic) matrice ET programmable, matrice OU figée),
- PLA (Programmable Logic Array) matrice ET ou matrice OU programmable,
- EPLD (Erasable PLD) effaçables par rayons ultraviolet, ils peuvent être reprogrammer,
- EEPLD (Electrically Erasable PLD) programmables et effaçables électriquement, ils peuvent être reprogrammés sur site. Les limites de l'architecture du PLD résident dans le nombre de bascules, le nombre de signaux d'entrées/sorties, la rigidité du plan logique ET OU et des interconnexions.
- FPGA (Field Programmable Gate Array) dont nous donnerons en détail l'explication ci-dessous.

4.2 FPGA (FIELD PROGRAMMABLE GATE ARRAYS)

FPGA se traduit en français par circuits prédiffusés programmables. C'est un composant standard combinant la densité et les performances d'un prédiffusé avec

la souplesse due à la reprogrammation des PLD. Cette configuration évite le passage chez le fondeur et tous les inconvénients qui en découlent.
Nous décrirons une famille de FPGA de Xilinx que nous avons utilisée. Il va de soi qu'il existe sur le marché nombre d'autres fondeurs qui ont développé des familles concurrentes ou complémentaires, parfois moins performantes, parfois mieux adaptées. Tout dépend aussi de l'utilisation que l'on en fait.

4.2.1 L'ARCHITECTURE DES CIRCUITS FPGA [VIRTEX-II][6]

Il existe actuellement plusieurs fabricants de circuits FPGA dont Xilinx et Altera sont les plus connus. L'architecture, retenue par Xilinx, se présente sous forme de deux couches :

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

La couche dite « circuit configurable » est constituée d'une matrice de blocs logiques configurables CLB permettant de réaliser des fonctions combinatoires et des fonctions séquentielles. Tout autour de ces blocs logiques configurables, nous trouvons des blocs entrées/sorties IOB dont le rôle est de gérer les entrées-sorties réalisant l'interface avec les modules extérieurs (figure 4.2). La programmation du circuit FPGA appelé aussi LCA (logic cells arrays) consiste à appliquer un potentiel adéquat sur la grille de certains transistors à effet de champ à interconnecter les éléments des CLB et des IOB afin de réaliser les fonctions souhaitées et d'assurer la propagation des signaux. Ces potentiels sont tout simplement mémorisés dans le réseau mémoire SRAM.

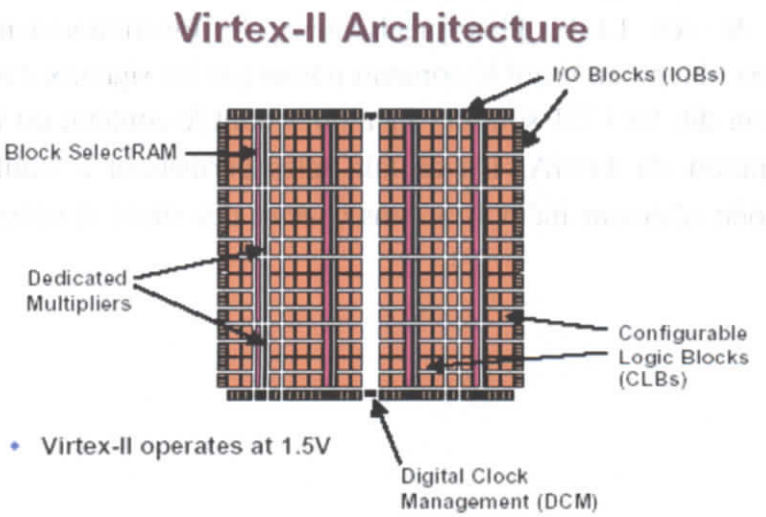


Figure 4.2 : Architecture interne d'un FPGA [7]

La configuration du circuit est mémorisée sur la couche réseau SRAM et stockée dans une ROM externe. Un dispositif interne permet à chaque mise sous tension de

charger la SRAM interne à partir de la ROM. Ainsi on conçoit aisément qu'un même circuit puisse être exploité successivement avec des ROM différentes puisque sa programmation interne n'est jamais définitive. On voit tout le parti que l'on peut tirer de cette souplesse en particulier lors d'une phase de mise au point. Une erreur n'est pas rédhibitoire, mais peut aisément être réparée. La mise au point d'une configuration s'effectue en deux temps: une première étape purement logicielle va consister à dessiner puis simuler logiquement le circuit fini, puis lorsque cette étape sera terminée on effectuera une simulation matérielle en configurant un circuit réel et l'on pourra alors vérifier si le fonctionnement réel correspond bien à l'attente du concepteur, et si besoin est identifier les anomalies liées généralement à des temps de transit réels légèrement différents de ceux supposés lors de la simulation logicielle ce qui peut conduire à des états instables voire même erronés..
Les circuits FPGA du fabricant Xilinx utilisent deux types de cellules de base :

- Les cellules d'entrées/sorties appelés IOB (input output bloc)
- Les cellules logiques appelées CLB (configurable logic bloc) Ces différentes cellules sont reliées entre elles par un réseau d'interconnexions configurable

LES CLB (CONFIGURABLE LOGIC BLOC)[8]

Les blocs logiques configurables sont les modules de base des circuits FPGA. Un CLB est constitué de quatre « slices ». La figure 4.3 illustre l'architecture simplifiée d'un CLB et celle d'un slice. La logique combinatoire est implantée grâce aux LUT (look-up tables) contenues dans chaque slice. Ces LUT peuvent également être configurées comme éléments de mémoire synchrone, simple ou double-port de 16 bits, ou encore comme registres à décalage de 16 bits. Il existe donc trois modes de configuration de ces LUT. Plus précisément, le fonctionnement en mode combinatoire est obtenu en lisant le contenu pointé par les signaux d'entrée (figure 4.4a). Autrement dit, les LUT sont des mémoires dont le contenu est initialisé lors de la configuration du FPGA. De ce fait, elles permettent à l'utilisateur d'en disposer en mode «élément mémoire» dans chacun des slices si nécessaire (figure 4.4b).

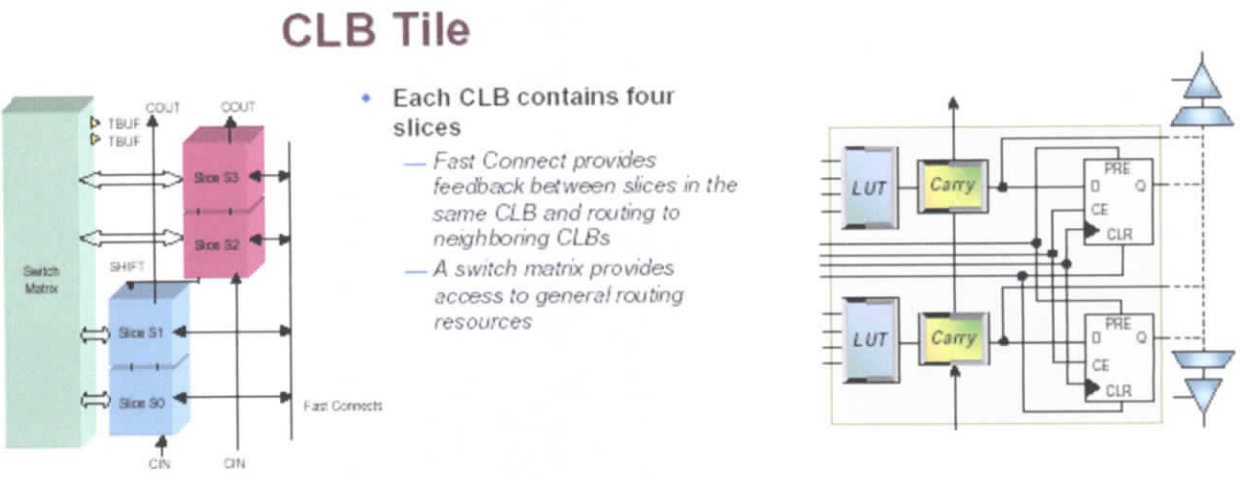


Figure 4.3 [8] : Architecture simplifiée :a) d'un CLB
b) d'un slice

La figure 4.4c décrit le mode de configuration particulier en registre à décalage de longueur programmable jusqu'à 16 bits.

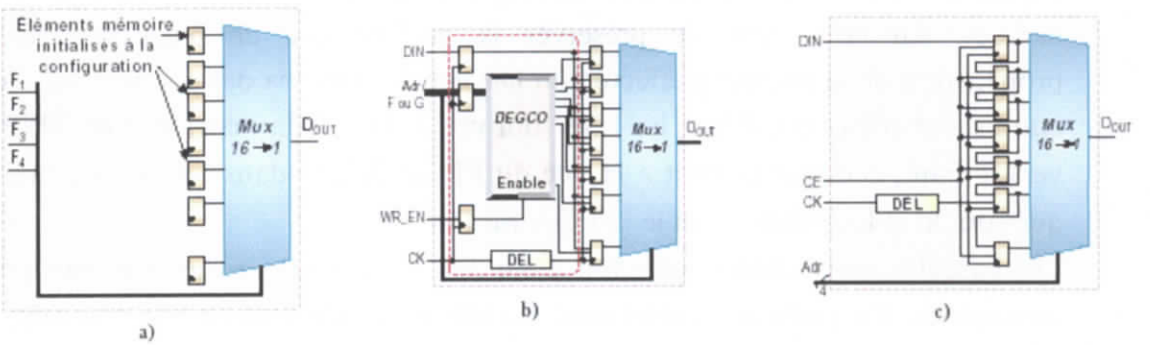


Figure 4.4 [8]
a) Configuration en fonction combinatoire.
b) Configuration en mémoire 16 bits à écriture synchrone
c) Configuration en registre à décalage de longueur programmable

Par ailleurs, une logique supplémentaire utile pour la réalisation de fonctions arithmétiques est disponible dans chaque slice. Grâce à ces éléments, et au style d'écriture adapté, des modules de type accumulateur en addition/soustraction pourront être implantés à raison de 2 bits par slice.

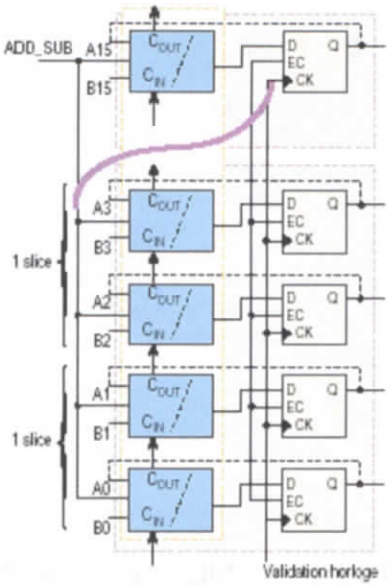


Figure 4.5 [8] :Une logique supplémentaire (Fast Carry)
permet l'implantation de fonctions de type
accumulateurs chargeables en addition/soustraction

A noter que l'appel à ces éléments de logique arithmétique, comme la propagation rapide de la retenue (figure 4.5), implique l'utilisation de ressources de routage dédiées. En effet, afin de préserver la performance de telles fonctions, la propagation de la retenue s'effectue du bas vers le haut via des connexions directes entre slices adjacents. Ainsi, les LSB sont vers le bas de la matrice et les MSB sont vers le haut, ceci par rapport à la vue du FPGA Xilinx dans les outils graphiques que sont le «Floorplanner» et le «FPGA Editor».

Les bascules dans chaque slice ont aussi des caractéristiques importantes pour le concepteur. En particulier, elles sont initialisées systématiquement à la mise sous tension (par défaut à la valeur '0'), et sont utilisables indépendamment de la logique combinatoire disponible dans le même slice. En outre, chaque bascule bénéficie de broches de contrôle telles que: entrée dédiée de validation de l'horloge (Clock enable) permettant d'activer ou de suspendre le fonctionnement de chacune des bascules individuellement, et ceci sans avoir à insérer de la logique combinatoire sur le chemin de l'horloge; entrées de «set» et de «reset» synchrones ou asynchrones. La polarité des signaux d'horloge, de Clock enable, de set et de reset est programmable pour chacune des bascules. Autrement dit, ces signaux peuvent être individuellement actifs au niveau haut ou au niveau bas.

Les outils de synthèse appropriés permettront de tirer profit de ces détails d'architecture à partir du code source VHDL (ou Verilog).

LES IOB (INPUT OUTPUT BLOC)[8]

Les blocs entrée/sortie permettent l'interface entre les broches du composant FPGA et la logique interne développée à l'intérieur du composant.

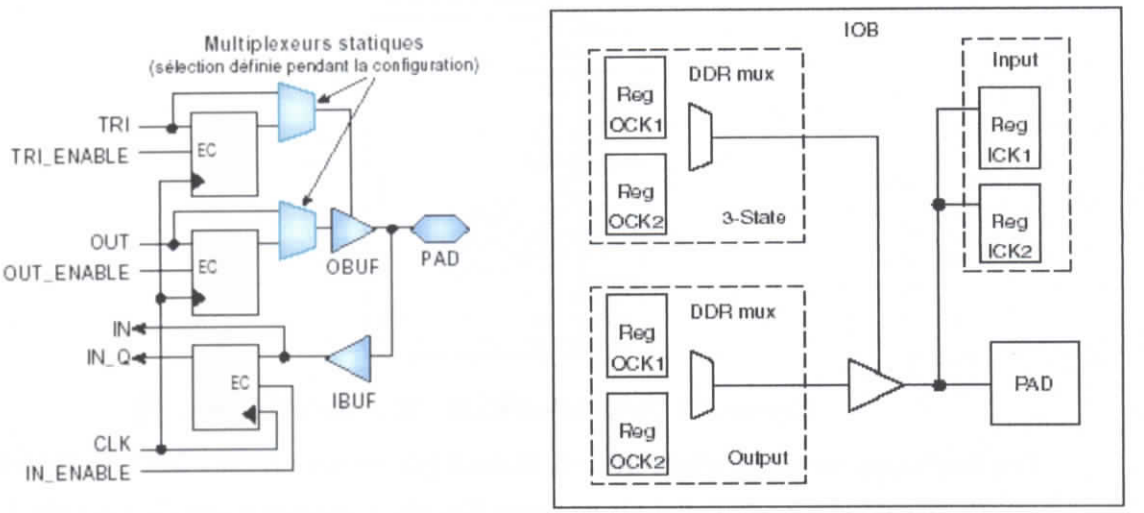


Figure 4.6 [8]: Bloc d'entrée/sortie (Virtex-II)

Ils sont présents sur toute la périphérie du circuit FPGA. Chaque bloc IOB contrôle une broche du composant et il peut être défini en entrée, en sortie, en signaux bidirectionnels ou être inutilisé (haute impédance).

L'échange de données avec la circuiterie externe (microprocesseur, DSP, mémoires rapides, convertisseurs A/N et N/A) est souvent un critère important dans la perspective des performances à atteindre. Les blocs d'entrées/sorties disposent de bascules sur les chemins d'entrée, de sortie et de contrôle trois états. Les outils de synthèse les plus sophistiqués permettent l'utilisation systématique (lorsque cela s'applique) de ces bascules, garantissant ainsi les temps d'établissement et de «clock-to-out» puisqu'ils deviennent indépendants du placement et donc du routage. La configuration électrique des blocs d'E/S donne la possibilité d'ajuster la raideur des fronts sur les étages de sortie, la sortance, les seuils de commutation et les standards de communication (LVTTL, LVCmos, SSTL, PCI, LVDS...). Par ailleurs, la famille Virtex-II offre d'adapter en impédance aussi bien les entrées que les sorties, sans avoir à insérer les traditionnelles résistances d'adaptation; le résultat est un gain de place important sur les circuits imprimés et une grande souplesse.

LES BLOCS MÉMOIRE DANS VIRTEX-II [8]

Les blocs «SelectRam» dans l'architecture Virtex-II sont des mémoires Ram double port de 18 Kbits (figure 4.7), configurables de différentes manières entre 18 K×1 bit et 512×36 bits. Chaque port est totalement synchrone et indépendant. Les blocs

«SelectRam» peuvent être mis en cascade pour réaliser de larges zones de stockage enfouies dans le FPGA. Un module multiplieur 18×18 bits est disposé à proximité de chaque bloc «Select-Ram» et est optimisé pour opérer sur le contenu d'un port de la mémoire.

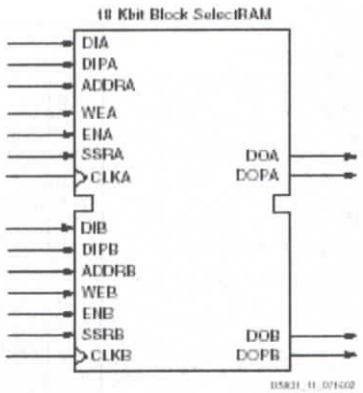


Figure 4.7 Bloc SelectRam 18Kbits double port[9]

Les fonctions mémoire «SelectRam» et multiplieur sont toutes deux connectées aux matrices d'interconnexions leur ouvrant l'accès aux ressources de routage globales de la matrice FPGA.

DISPOSITIFS DE GESTION DES HORLOGES (DIGITAL CLOCK MANAGEMENT :DCM)[8]

La plupart des FPGA, et notamment ceux de Xilinx, disposent de ressources (buffers et routage associé) qui permettent une répartition d'horloge parfaite, c'est-à-dire sans dérive (skew); un point particulièrement important pour les conceptions synchrones.

Le déroulement des événements est le suivant:

Les outils de synthèse détectent automatiquement les sources d'horloge, et infèrent les buffers spécialisés et les ressources de routage associées (figure 4.8 a). Les dispositifs de gestion des horloges permettent l'implantation aisée et efficace de fonctions comme la multiplication et la division de la fréquence.

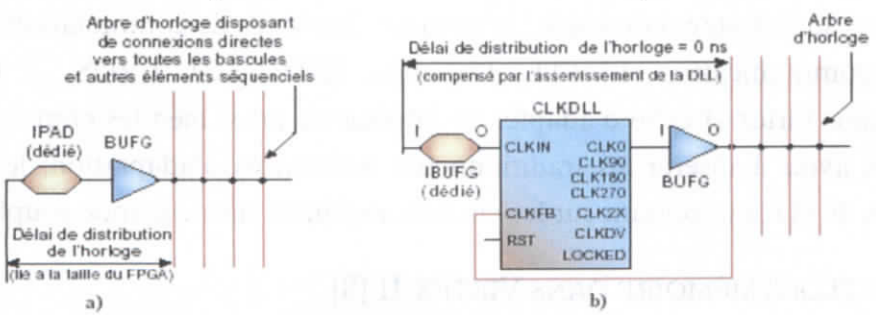


Figure 4.8 [8] : a) Les buffers d'horloge et les ressources de routages associées
b) Des dispositifs de gestion des horloges permettant d'adapter la fréquence d'horloge

4.2.2 CARACTÉRISTIQUES DES FPGA DE XILINX [8]

Les différentes architectures FPGA Xilinx sont Spartan-II, Spartan-IIE, Virtex-E et Virtex-II. Leurs caractéristiques principales sont:

- Complexités allant de 15000 à plus de 8 millions de portes.
- Faible consommation.
- Grande souplesse d'utilisation des entrées-sorties avec adaptation d'impédance (Virtex-II).
- Fonctions mémoire (distribuée et blocs de Ram).
- Dispositifs de gestion des horloges (DCM).
- Multiplieurs câblés (Virtex-II). Et bien d'autres possibilités permettant d'optimiser à la fois la performance et la densité des fonctions logiques et/ou arithmétiques.

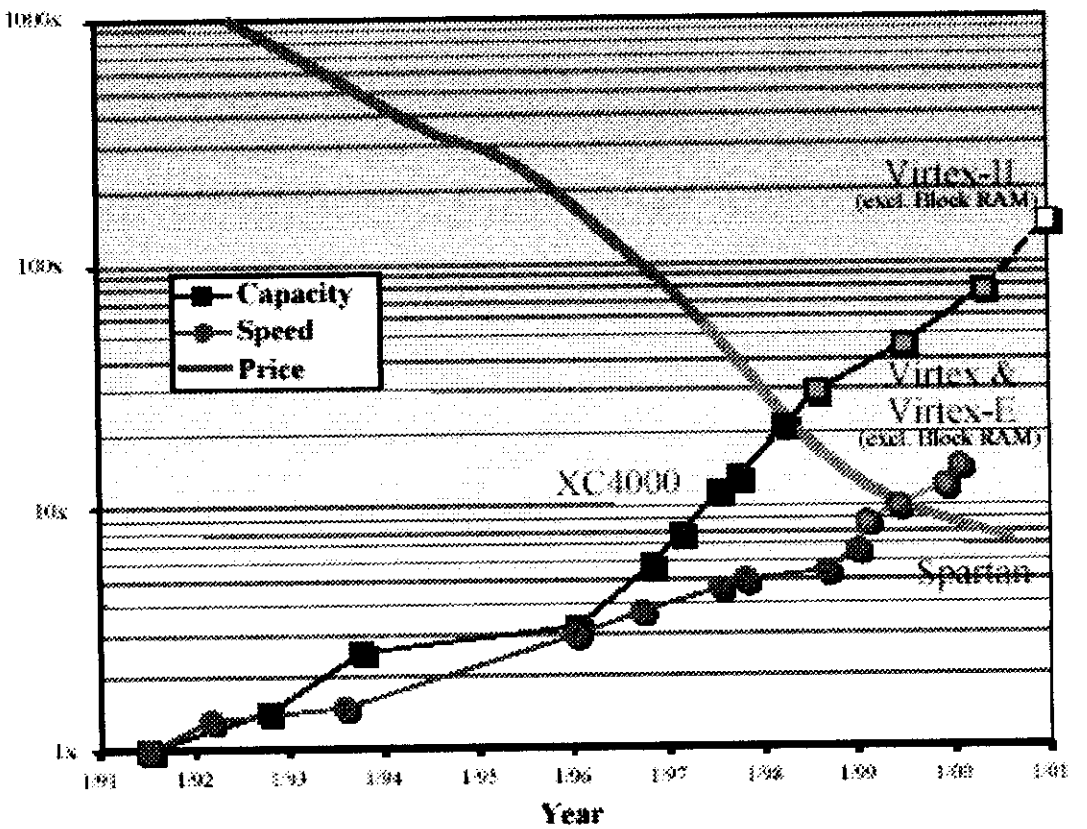


Figure 4.9 Progrès récents des FPGA Xilinx [7]

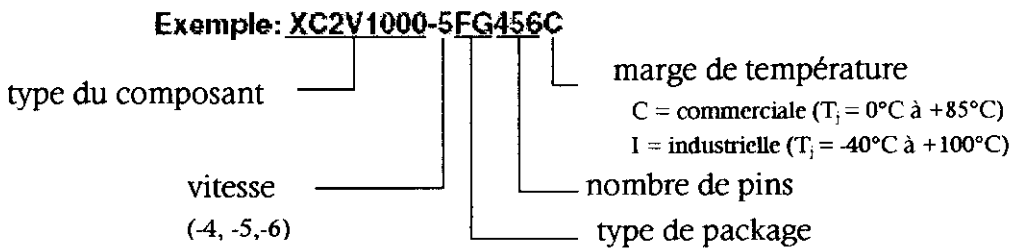
4.2.3 CARACTÉRISTIQUES GENERALES DE LA FAMILLE VIRTEX-II [9]

Table 1: Virtex-II Field-Programmable Gate Array Family Members

Device	System Gates	CLB (1 CLB = 4 slices = Max 128 bits)			Multiplier Blocks	SelectRAM Blocks		DCMs	Max I/O Pads[1]
		Array Row x Col.	Slices	Maximum Distributed RAM Kbits		18 Kbit Blocks	Max RAM (Kbits)		
XC2V40	40K	8 x 8	256	8	4	4	72	4	88
XC2V80	80K	16 x 8	512	16	8	8	144	4	120
XC2V250	250K	24 x 16	1,536	48	24	24	432	8	200
XC2V500	500K	32 x 24	3,072	96	32	32	576	8	264
XC2V1000	1M	40 x 32	5,120	160	40	40	720	8	432
XC2V1500	1.5M	48 x 40	7,680	240	48	48	864	8	528
XC2V2000	2M	56 x 48	10,752	336	56	56	1,008	8	624
XC2V3000	3M	64 x 56	14,336	448	96	96	1,728	12	720
XC2V4000	4M	80 x 72	23,040	720	120	120	2,160	12	912
XC2V6000	6M	96 x 88	33,792	1,056	144	144	2,592	12	1,104
XC2V8000	8M	112 x 104	46,592	1,456	168	168	3,024	12	1,108

Tableau 4.1 :Caractéristique de la famille Virtex-II

4.2.4 RÉFÉRENCE DES FPGA VIRTEX-II [9]



4.2.5 AVANTAGES DES CIRCUITS FPGA

- Le premier argument est la souplesse de programmation qui permet l'emploi conjoint d'outils de saisie de schéma et l'exploitation d'un langage de haut niveau tel que le VHDL. Ce qui permet de multiplier les essais, d'optimiser de diverses manières l'architecture développée, de vérifier à divers niveaux de simulation la fonctionnalité de cette architecture.
- Le second argument est évidemment la nouvelle possibilité de reconfiguration dynamique partielle ou totale d'un circuit ce qui permet d'une part, une meilleure exploitation du composant, une réduction de surface de silicium employé et donc du coût, et d'autre part, une évolutivité assurant la possibilité de couvrir à terme des besoins nouveaux sans nécessairement repenser l'architecture dans sa totalité. L'un des points forts de la reconfiguration dynamique est effectivement de permettre de reconfigurer en temps réel en quelques microsecondes tout ou partie du

circuit, c'est à dire de permettre de modifier la fonctionnalité d'un circuit en temps quasi réel. Ainsi le même CLB pourra à un instant donné être intégré dans un processus de filtrage numérique d'un signal et l'instant d'après être utilisé pour gérer une alarme. On dispose donc quasiment de la souplesse d'un système informatique qui peut exploiter successivement des programmes différents, mais avec la différence fondamentale qu'ici il ne s'agit pas de logiciel mais de configuration matérielle, ce qui est infiniment plus puissant.

- Notons enfin que ces circuits n'ont pas vocation à concurrencer les super calculateurs, mais plutôt à offrir une alternative en fonction de critères comme l'encombrement, les performances et le prix, et sont de ce fait bien adaptés à des applications de qualité dans le domaine des systèmes versatiles.

4.3 MÉTHODOLOGIE DE CONCEPTION TOP-DOWN

Pour la programmation de l'FPGA, on utilise le cycle Foundation (figure 4.10) de la compagnie Xilinx qui est le leader actuellement dans le domaine des FPGAs.

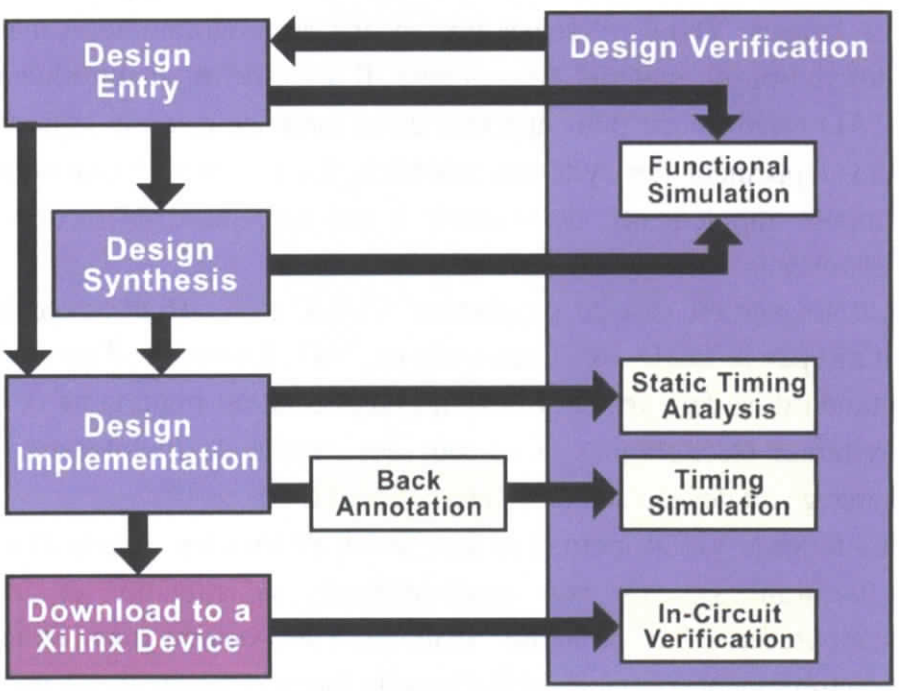


Figure 4.10 : Cycle de développement [7]

Ce cycle est composé principalement de cinq (05) modules :

- **Design entry**: module permettant la saisie de l'architecture qu'elle soit décrite graphiquement ou textuellement (décrite dans un langage de description hardware, VHDL ou Verilog).

- Design Synthesis : module permettant de traduire l'architecture en terme de portes logiques (obtention d'une "netlist" optimisée). Le synthétiseur va reconnaître un certain nombre de primitives qu'il va implanter et connecter entre elles.
- Design Implementation: module permettant de faire le partitionement, placement et routage de la "netlist" optimisée selon le circuit FPGA sélectionné.
- Design Verification: Module permettant de faire les vérifications nécessaires pour s'assurer du bon déroulement des opérations de saisie, de synthèse et d'implémentation. Celui ci consiste en une simulation fonctionnelle et une simulation temporelle, c'est à dire après placement et routage.
- Download to Xilinx Device : Une fois l'implémentation validée par les différentes simulations, ce module permet de configurer le circuit FPGA en injectant par le biais d'un programmateur un fichier binaire à l'intérieur du circuit.

4.4 LE LANGAGE DE DESCRIPTION DE MATERIEL VHDL

Le langage VHDL n'est pas un langage de programmation mais un langage de description de matériel électronique. Il a été défini et introduit dans les outils de CAO électronique pour apporter de la méthode et de la rigueur dans le cycle de développement des systèmes matériels. La complexité croissante des systèmes a imposé rapidement de recourir à des niveaux d'abstraction de plus en plus importants.

Ce langage est issu du programme VHSIC (Very High Speed Integrated Circuit) initie par le DOD aux Etats-Unis en 1981. Le besoin d'un langage moderne et standard, se fait ressentir devant l'augmentation importante de la complexité des systèmes électroniques et surtout des coûts de maintenance en résultant. Ce langage est devenu standard international IEEE en 1987.

Le langage VHDL permet la description de tous les aspects d'un système matériel (*hardware system*): son comportement, sa structure et ses caractéristiques temporelles. Par système matériel, on entend un système électronique arbitrairement complexe réalisé sous la forme d'un circuit intégré ou d'un ensemble de cartes. Le comportement définit la ou les fonctions que le système remplit (par exemple le comportement d'un microprocesseur comporte, entre autres, des fonctions arithmétiques et logiques). La structure définit l'organisation du système en une hiérarchie de composants (par exemple un microprocesseur est constitué d'une unité de contrôle et d'une unité opérative; cette dernière est elle-même, entre autres, constituée d'un composant réalisant les opérations arithmétiques entières et d'un composant réalisant les opérations arithmétiques en virgule flottante). Les

CHAPITRE V

ARCHITECTURES PROPOSÉES

Dans ce présent chapitre, nous allons présenter les architectures qui résolvent le problème du chemin algébrique à savoir la fermeture transitive d'un graphe, la longueur du chemin le plus court entre deux sommets d'un graphe et l'arbre couvrant de poids minimum. Ces architectures traduisent les algorithmes que nous avons vus dans le deuxième chapitre. Du fait que le flot de données est le même dans les trois algorithmes, les architectures auront la même structure générale; la différence résidera dans les processeurs élémentaires, car les opérations de base sont propres à chaque algorithme.

5.1 STRUCTURE GÉNÉRALE

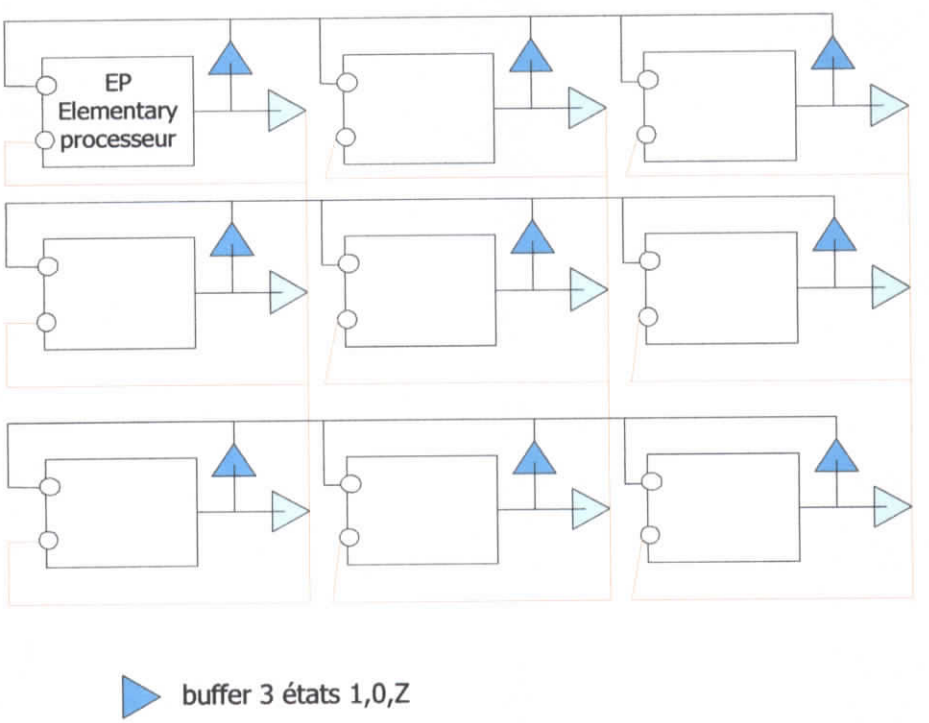


Figure 5.1 : Exemple de l'architecture générale pour une matrice 3×3

Nous avons vu au chapitre 2 que durant la $k^{ième}$ étape de chaque algorithme les données se trouvant sur la $k^{ième}$ ligne et la $k^{ième}$ colonne étaient distribuées vers leurs colonnes et leurs lignes respectives. Prenons à titre d'exemple la 3ème étape d'un des algorithmes :

$$a_{ij}^{(3)} = a_{ij}^{(2)} \oplus a_{i3}^{(2)} \otimes a_{3j}^{(2)}$$

caractéristiques temporelles définissent des contraintes sur le comportement du système (par exemple les signaux d'un bus de données doivent être stables depuis un temps minimum donné par rapport à un front d'horloge pour qu'une opération d'écriture dans la mémoire soit valable).

4.5 CONCLUSION

Nous avons passé en revue dans ce chapitre quelques éléments de technologie et donner les raisons qui nous ont conduit à choisir les FPGA. Nous avons ensuite décrit les spécificités de la famille Virtex-II; famille que nous utilisons dans ce projet. Enfin nous avons décrit la méthodologie de conception top-down que nous avons adopté dans ce travail ainsi que le langage VHDL pour la description de circuits.

Où $\oplus$ et $\otimes$ désignent les opérations élémentaires propres à chaque algorithme.

Le calcule de $a_{ij}^{(3)}$ se fait dans le processeur élémentaire qui se trouve sur la ligne i, et la colonne j (on le désignera par Ep (i,j)). Pour cela il utilisera la donnée $a_{ij}^{(2)}$ calculée lors de l'étape précédente, présente dans Ep (i,j), et les données $a_{i3}^{(2)}$, $a_{3j}^{(2)}$ présentes respectivement dans Ep (3,j), Ep (i,3). On aura donc à propager les données qui se trouvent sur la 3ème ligne, et la 3ème colonne respectivement vers leurs colonnes et leurs lignes respectives, c'est ainsi que l'élément a_{23} sera distribué vers la 2ème ligne, et l'élément a_{34} sera distribué vers la 4ème colonne.

Chaque Ep (i,j) sera relié par deux buffers 3 états à la ligne i et la colonne j. On commandera ces buffers lorsqu'il faudra propager la donnée qui se trouve dans Ep (i,j). Ceci explique la structure générale présentée sur la figure 5.1. Nous allons maintenant présenter les processeurs élémentaires de chaque algorithme.

5.2 PROCESSEUR ÉLÉMENTAIRE POUR LA FERMETURE TRANSITIVE

Le type de donnée traitée dans l'algorithme de la fermeture transitive est le type binaire, et les opérations élémentaires sont le ET logique, et le OU logique. L'expression de récurrence est :

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} OU (a_{ik}^{(k-1)} ET a_{kj}^{(k-1)})$$

Où k désigne la k ième phase de calcul. On notera $a_{ij}^{(0)}$ les données initiales.

On a donc besoin pour le calcul de $a_{ij}^{(k)}$:

- d'un élément de mémoire qui gardera la valeur de $a_{ij}^{(k-1)}$.
- d'une porte ET.
- d'une porte OU.

Voici donc un schéma de principe du processeur élémentaire.

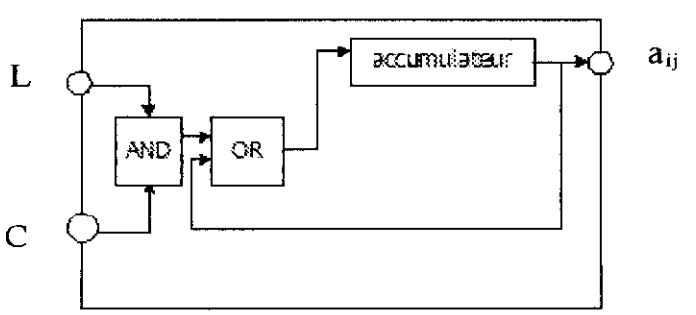


Figure 5.2 Schéma de principe du processeur élémentaire pour l'algorithme de la fermeture transitive

L'entrée L (ligne) recevra la donnée $a_{ik}^{(k-1)}$ qui se trouve sur la même ligne que a_{ij}, et l'entrée C (colonne) recevra la donnée $a_{kj}^{(k-1)}$ qui se trouve sur la même colonne que

a_{ij}. Vu que l'aspect synchronisme est primordial dans une architecture systolique, l'élément de mémoire sera une bascule D qui sera pilotée par l'horloge.

Il faut penser maintenant au chargement de la matrice. Il est clair que les entrée L et C ne peuvent servir pour ceci; on est donc amener à utiliser une entrée supplémentaire Ain (INPUT) qui devra avoir accès directement à la bascule D (élément de mémoire). Le chargement de la matrice se fera ligne par ligne (ou colonne par colonne) en commençant par la dernière ligne (la dernière colonne). Il faudra séparer le chemin de données de chargement, du chemin de données de traitement ; et pour ce faire on utilisera un multiplexeur. Ceci nous conduit à un deuxième schéma plus élaboré (figure 5.3)

L'entrée SMUX (Select Multiplexer) sera mise à '1' pendant le chargement, puis à '0' pendant le calcul. Du moment que chaque processeur élémentaire aura à sa sortie deux buffers 3 états, on peut les intégrer dans l'EP(figure 5.4), ensuite on aura plus qu'à former une matrice d' EP pour avoir l'architecture globale.

Si on envisage un chargement colonne par colonne, on aura à connecter la sortie Aout de l'EP(i,j) à l'entrée Ain de l'EP(i,j+1). La sortie OL (Output Line) sera connectée à la ligne commune aux EP (i,1:N), quand à la sortie OC (Output Column), elle sera connectée à la ligne commune aux EP(1:N,j).

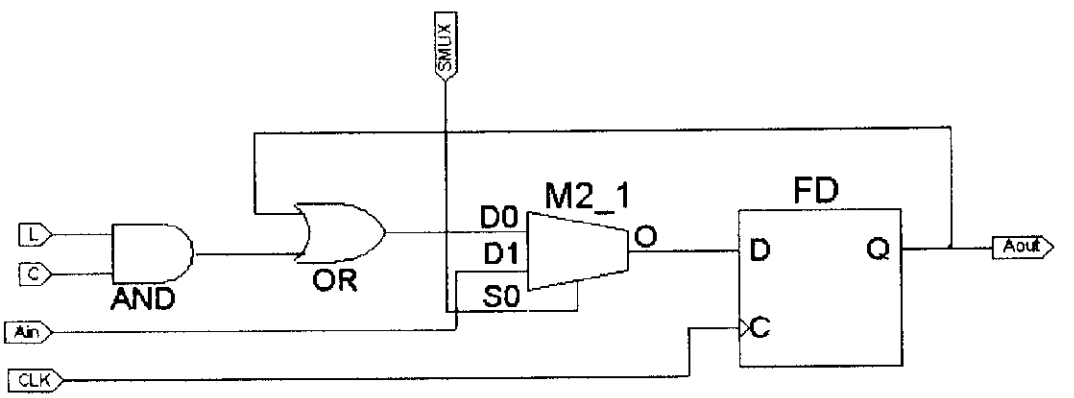


Figure 5.3: Processeur élémentaire modifié

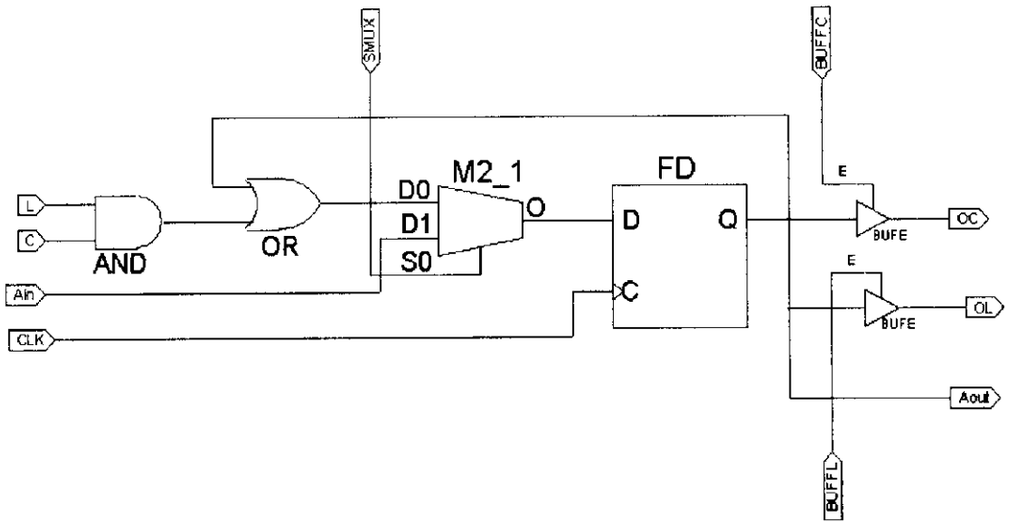


Figure 5.4 : Version finale du processeur élémentaire

5.3 PROCESSEUR ÉLÉMENTAIRE POUR LA LONGUEUR DU CHEMIN LE PLUS COURT

Le type de données utilisées est le type entier, il sera codé sur 4, 8, 16 bits. L'expression de récurrence pour cet algorithme est :

$$a_{ij}^{(k)} = \min\left(a_{ij}^{(k-1)}, \left[a_{ik}^{(k-1)} + a_{kj}^{(k-1)}\right]\right)$$

En adoptant la même démarche que précédemment on aboutit au schéma suivant.

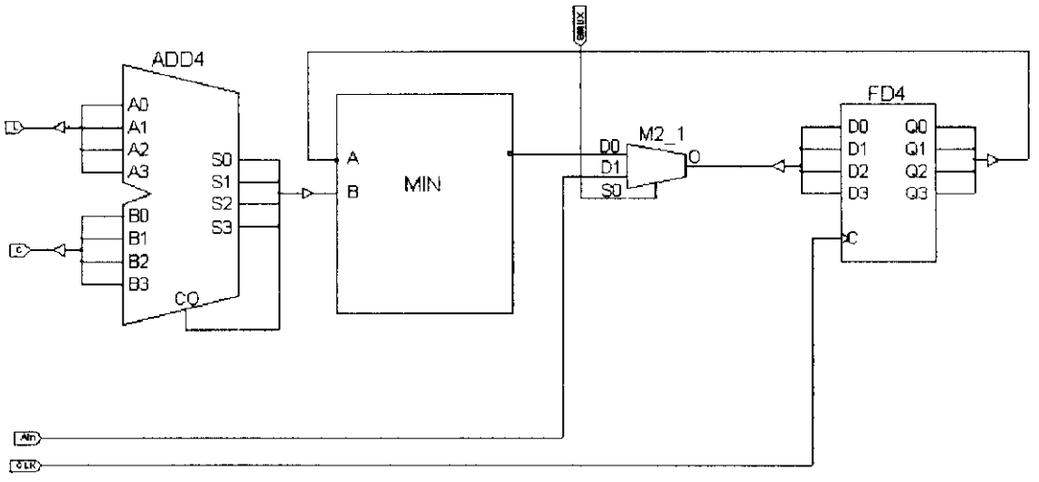


Figure 5.5 Processeur élémentaire pour l'algorithme de calcul de la longueur du chemin le plus court (taille de la donnée =4)

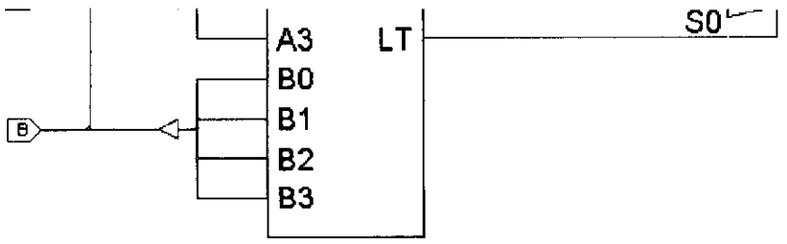


Figure 5.9: Détail de la fonction MAX.

La fonction MIN est réalisée en utilisant un comparateur et un multiplexeur en plus d'une porte AND à une entrée inversée pour prendre en compte la retenue (CO : Carry Out). Le principe de fonctionnement de la fonction MIN est comme suit :
GT = 1 quand A > B, 0 sinon. Lorsque S0 = 1 O <= D0, lorsque S0 = 0 O <= D1
Si la somme ne contient pas de retenu (CO=0) l'entrée inversée de la porte AND reçoit 0 inhibant de ce fait la porte AND (S0 = GT) ; si A > B alors GT = S0 = 1, la sortie du multiplexeur reçoit B (le minimum des deux entrée A et B) ; si A < B alors GT = S0 = 0, la sortie du multiplexeur reçoit A (le minimum des deux entrée A et B). Si CO = 1 (présence de retenu et donc B > A), S0 = 0, la sortie du multiplexeur

5.5 LA MATRICE D'EP

Comme on vient de voir plus haut, la structure générale est une matrice de processeurs élémentaires, liés localement entre eux. Cette architecture possède :

- un port d'entrée (INPUT) qui sert à charger la matrice à traiter, colonne par colonne (ou ligne par ligne).
- un port de sortie (OUTPUT) qui sert à visualiser les résultats, colonne par colonne (ou ligne par ligne).
- un port SMUX (Select Multiplexer) pour piloter les multiplexeurs contenus dans chaque EP.
- un port SBUFF (Select Buffer) pour piloter les buffers, afin de propager les données durant chaque phase de calcul.

5.6 LE CONTRÔLEUR

Une fois la description des processeurs élémentaires et la structure générale terminées, il va falloir piloter le tout. C'est le rôle du contrôleur. Nous commençons par établir son cahier des charges.
Il a deux entrées : RESET pour la remise à zéro du compteur et des bascules qu'il contient, et CLOCK comme entrée d'horloge.
Il doit générer deux signaux :

- SMUX qui pilotera les multiplexeurs, Il est mis à 1 pendant les N premiers cycles d'horloge (chargement), puis à zéro pendant les N cycles restants (traitement).
- SBUFF (Bus de N fils) qui pilotera les buffers. Pendant la phase de traitement, il validera une ligne et une colonne seulement pendant chaque cycle d'horloge.

On le réalisera en utilisant le formalisme machine d'état. Cette dernière possède deux états: LOAD pour le chargement et TREAT pour le traitement. Elle possède aussi un signal interne dénommé COUNT (Compteur) afin que chaque état dure N cycles[5] d'horloge.

[5] N taille de la matrice à piloter.

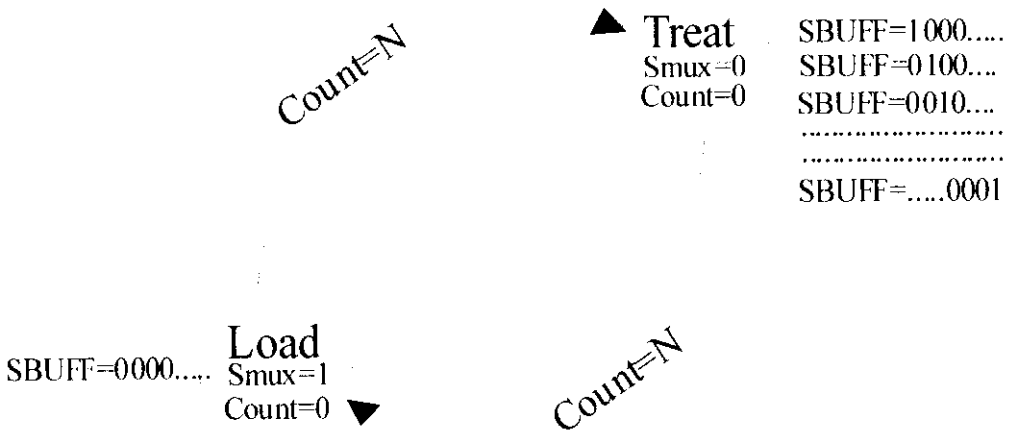


Figure 5.10 : Graphe de la machine d'état du contrôleur

5.7 CARACTÉRISTIQUES DES ARCHITECTURES

- La première caractéristique de ces architectures est le temps d'éxécution qui est de N cycles d'horloges, ce qui à notre connaissance n'a pas encore été atteint auparavant.
- Ces architectures sont des architectures semi-systoliques
- Le gain en vitesse est de $S=N^3/N=N^2$, car le temps d'exécution de l'algorithme de Warshall-Floyd est de N^3 (trois boucles de longueur N). Ce gain est atteint en utilisant N^2 cellules élémentaires, d'où une efficacité de $E=N^2/N^2=1$. Ceci signifie que le parallélisme a été poussé au maximum.
- La latency est de N car $L_{de}=N-1$ (le temps qui sépare l'entrée de la première donnée de celle de la dernière donnée), $L_{dd}=0$ (il n'y a pas de délai avant l'entrée de la première donnée), $L_{dp}=1$ car chaque donnée est présente simultanément sur toutes les cellules de sa ligne /colonne.
- Les architectures nécessitent un préchargement des données.

5.8 CONCLUSION

Nous avons exposé dans ce chapitre les architectures qui résolvent le problème du chemin algébrique. Pour vérifier qu'elles remplissent bien le cahier des charges que nous nous sommes fixés au départ, à savoir un temps d'exécution de N cycles d'horloge, nous allons procéder à des tests dont les résultats seront exposés dans le chapitre suivant.

CHAPITRE VI

IMPLÉMENTATION POUR LE 2V8000

Dans ce chapitre, nous testons les architectures proposées au chapitre 5 au niveau comportemental et post-placement et routage (à titre indicatif pour la taille 6×6). On donnera aussi les rapports de synthèse. Ces rapports constituent un inventaire des ressources utilisées par les architectures. Enfin nous montrerons les limites des tests manuels et donnerons comme alternative un test automatique.

6.1 TEST DE L'ARCHITECTURE DE LA FERMETURE TRANSITIVE

La figure ci dessus montre les chronogrammes obtenus lors du test fonctionnel :

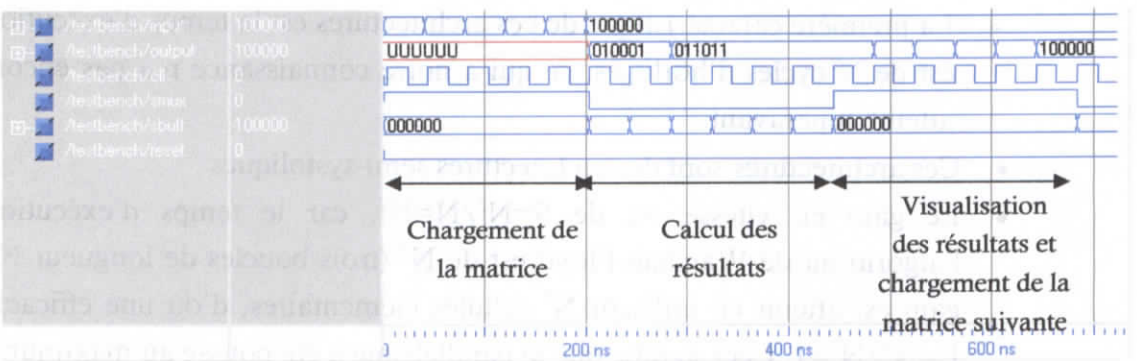


Figure 6.1 : Test du modèle comportementale de la TS

On observe une phase de chargement de N cycles et une phase de traitement de N cycles aussi. Ceci correspond bien au cahier des charges fixé au départ. La figure suivante montrent les chronogrammes obtenus lors du test post-placement et routage du circuit en utilisant un modèle du circuit qui inclue les délais de chaque élément et les délais de routage.

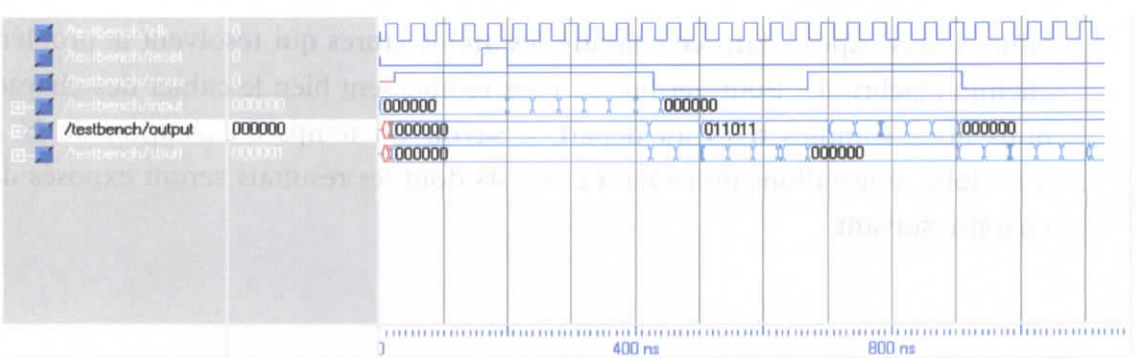


Figure 6.2: Test post-placement et routage de la TS

On observe que le circuit remplit le cahier des charges et que les différents délais de routage et des niveaux logiques n'entraînent pas un fonctionnement erroné. Un zoom (figure 6.3) permet d'observer l'effet des états intermédiaires et des retards entre la sortie et l'horloge, chose que le test fonctionnel ne fait pas ressortir.

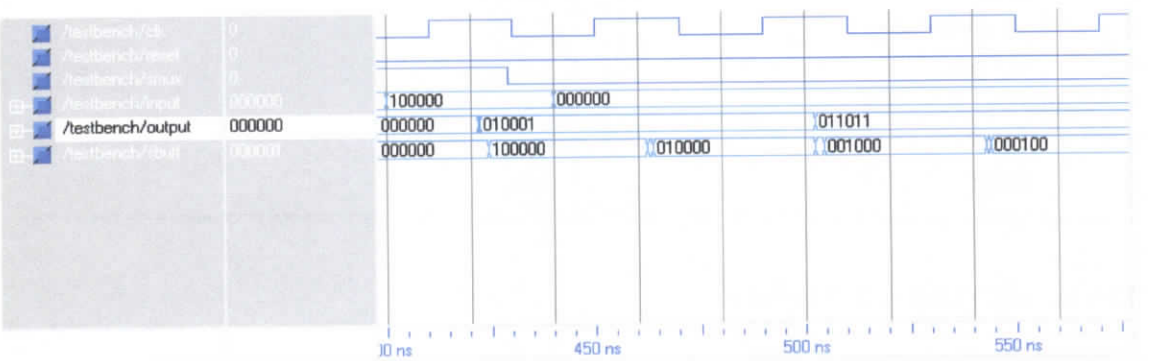


Figure 6.3 : Effet des délais.

La figure suivante présente le rapport de synthèse :

```
================================================================================
*                           Final Report                                       *
================================================================================
Final Results
RTL Output File Name                : archi.ngr
Top Level Output File Name          : archi
Output Format                       : NGC
Optimization Criterion              : Speed
Keep Hierarchy                      : NO
Macro Generator                     : macro+

Design Statistics
# IOs                               : 21

Macro Statistics :
# Registers                         : 38
#      1-bit register               : 37
#      3-bit register               : 1

Cell Usage :
# BELS                              : 120
#      GND                          : 36
#      LUT1                         : 1
#      LUT1_L                       : 1
#      LUT2                         : 1
#      LUT3                         : 1
#      LUT3_L                       : 36
#      LUT4                         : 38
#      LUT4_L                       : 6
```

```
# FlipFlops/Latches                 : 40
#      FD                           : 36
#      FDC                          : 1
#      FDR                          : 3
# Tri-States                        : 72
#      bufe                         : 72
# Clock Buffers                     : 1
#      BUFGP                        : 1
# IO Buffers                        : 20
#      IBUF                         : 7
#      OBUF                         : 13
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2v8000bf957-5

 Number of Slices:                      79  out of  46592     0%
 Number of Slice Flip Flops:            40  out of  93184     0%
 Number of 4 input LUTs:                84  out of  93184     0%
 Number of bonded IOBs:                 20  out of    684     2%
 Number of TBUFs:                       72  out of  23296     0%
 Number of GCLKs:                        1  out of     16     6%
```

Figure 6.4 Rapport de synthèse de la TS

Les ressources utilisées représentent environ 6% des ressources globales, on pourrait prévoir que la taille maximum qui puissent être implémentée sur ce circuit (2V8000bf957-5) est 100×100.

6.2 TEST DE L'ARCHITECTURE LA LONGUEUR DU CHEMIN LE PLUS COURT

La figure ci dessus montre les chronogrammes obtenus lors du test fonctionnel :

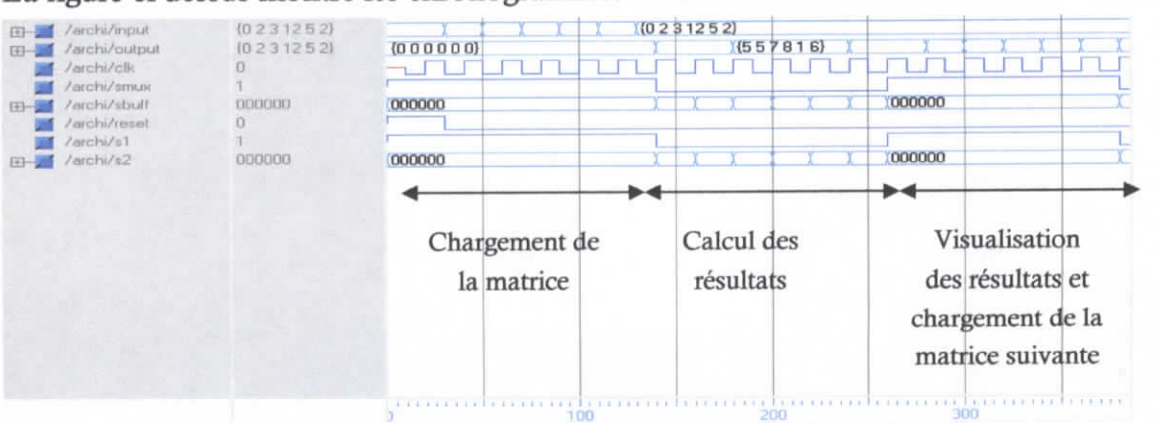


Figure 6.5 : Test du modèle comportementale de la SP

Ici aussi le test est satisfaisant (taille de la donnée = 4 bits), on observe une phase de chargement en N cycles et une phase de traitement en N cycles.
La figure suivante montre les chronogrammes obtenus lors du test post-placement et routage.

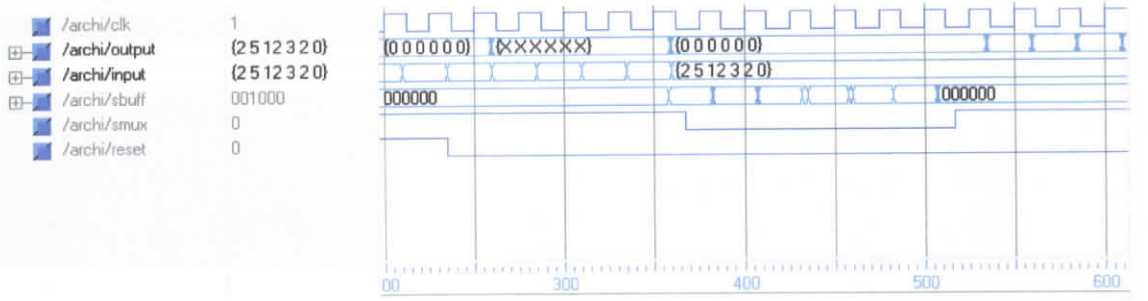


Figure 6.6 : Test post-placement et routage de la SP

Les différents délais n'entraînent pas un fonctionnement erroné.
La figure suivante montre l'effet de ces délais.

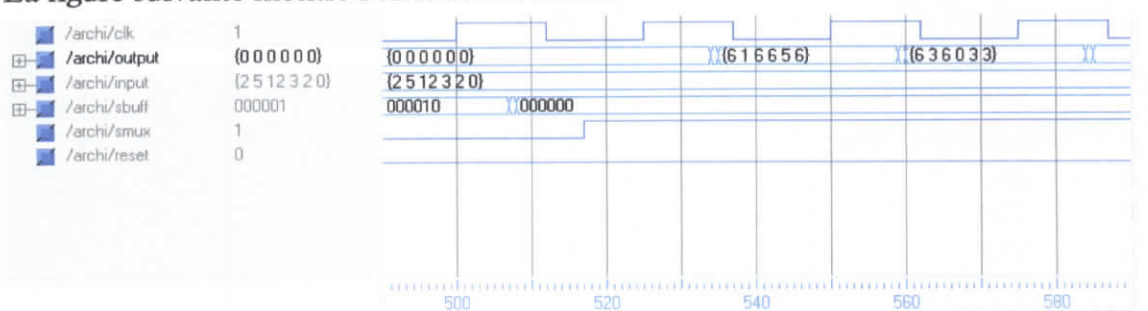


Figure 6.7 : Effet des délais.

La figure suivante présente le rapport de synthèse

```
=========================================================================
*                            Final Report                               *
=========================================================================
Final Results
RTL Output File Name                : archi.ngr
Top Level Output File Name          : archi
Output Format                       : NGC
Optimization Criterion              : Speed
Keep Hierarchy                      : NO
Macro Generator                     : macro+

Design Statistics
# IOs                               : 57

Macro Statistics :
# Registers                         : 38
#      1-bit register               : 1
#      3-bit register               : 1
#      4-bit register               : 36
```

```
# Adders/Subtractors                    : 36
#      4-bit adder carry out            : 36
# Comparators                           : 36
#      4-bit comparator greater         : 36

Cell Usage :
# BELS                                  : 1136
#      and2b1                           : 36
#      buf                              : 7
#      GND                              : 1
#      LUT1                             : 2
#      LUT2                             : 109
#      LUT2_D                           : 36
#      LUT2_L                           : 108
#      LUT3                             : 289
#      LUT4                             : 110
#      LUT4_L                           : 42
#      MUXCY                            : 144
#      MUXF5                            : 144
#      XORCY                            : 108
# FlipFlops/Latches                     : 148
#      FD                               : 144
#      FDC                              : 1
#      FDR                              : 3
# Tri-States                            : 288
#      bufe                             : 288
# Clock Buffers                         : 1
#      bufg                             : 1
# IO Buffers                            : 57
#      IBUF                             : 25
#      IBUFG                            : 1
#      OBUF                             : 31
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2v8000bf957-5

 Number of Slices:                      362  out of  46592     0%
 Number of Slice Flip Flops:            148  out of  93184     0%
 Number of 4 input LUTs:                696  out of  93184     0%
 Number of bonded IOBs:                  57  out of    684     8%
 Number of TBUFs:                       288  out of  23296     1%
 Number of GCLKs:                         1  out of     16     6%
```

Figure 6.8: Rapport de synthèse de la SP

Les ressources utilisées se situent autour de 8%, la taille maximum qu'on peut prévoir est de 75×75.

6.3 SYNTHESE DE L'ARCHITECTURE DE L'ARBRE COUVRANT DE POIDS MINIMUM

La figure ci-dessous montre les chronogrammes obtenus lors du test fonctionnel.

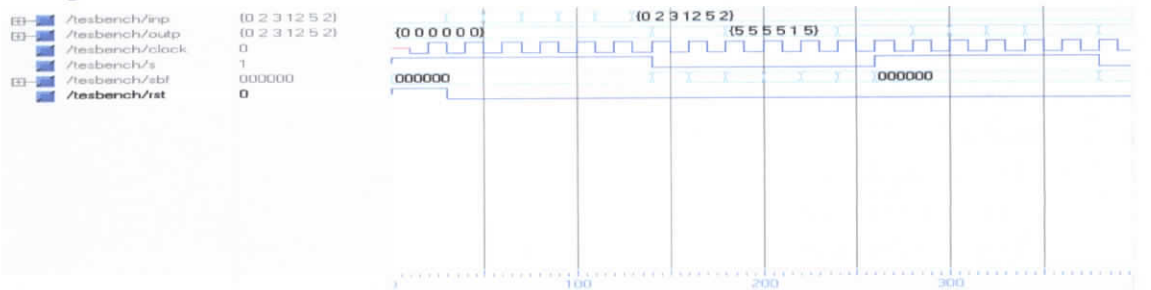


Figure 6.9 : Test fonctionnel de la MWST

Le fonctionnement est conforme au cahier des charges.

La figure suivante montre le test post-placement et routage.

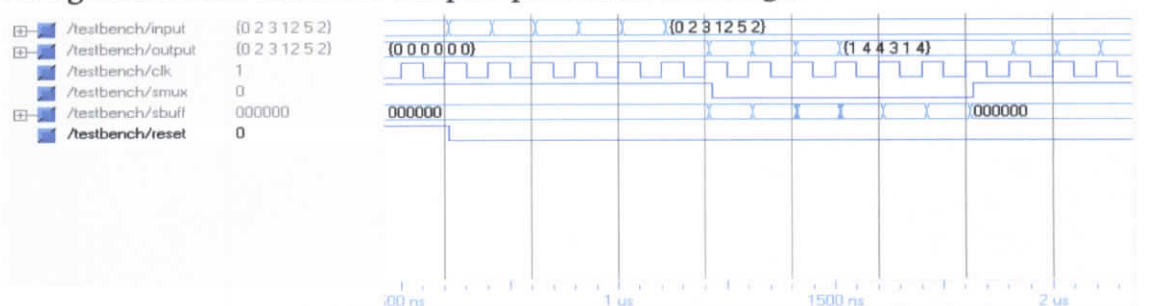


Figure 6.10 : Test post-placement et routage de la MWST

Les différents délais n'entraînent pas un fonctionnement erroné.

La figure suivante montre l'effet des délais.

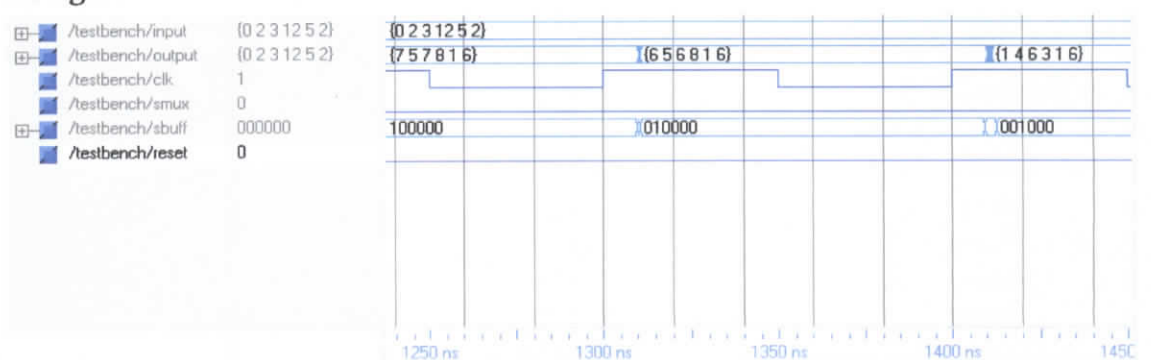


Figure 6.11 : Effet des délais

La figure suivante présente le rapport de synthèse

```
================================================================================
*                              Final Report                                    *
================================================================================
Final Results
```

```
RTL Output File Name                : archi.ngr
Top Level Output File Name          : archi
Output Format                       : NGC
Optimization Criterion              : Speed
Keep Hierarchy                      : NO
Macro Generator                     : macro+

Design Statistics
# IOs                               : 57

Macro Statistics :
# Registers                         : 38
#      1-bit register               : 1
#      3-bit register               : 1
#      4-bit register               : 36
# Comparators                       : 72
#      4-bit comparator greater     : 36
#      4-bit comparator less        : 36

Cell Usage :
# BELS                              : 1027
#      buf                          : 7
#      LUT1                         : 2
#      LUT2                         : 145
#      LUT3                         : 433
#      LUT3_D                       : 72
#      LUT4                         : 146
#      LUT4_L                       : 78
#      MUXF5                        : 144
# FlipFlops/Latches                 : 148
#      FD                           : 144
#      FDC                          : 1
#      FDR                          : 3
# Tri-States                        : 288
#      bufe                         : 288
# Clock Buffers                     : 1
#      bufg                         : 1
# IO Buffers                        : 57
#      IBUF                         : 25
#      IBUFG                        : 1
#      OBUF                         : 31
=========================================================================

Device utilization summary:
---------------------------

Selected Device : 2v8000bf957-5
```

```
Number of Slices:                461  out of  46592     0%
Number of Slice Flip Flops:      148  out of  93184     0%
Number of 4 input LUTs:          876  out of  93184     0%
Number of bonded IOBs:            57  out of    684     8%
Number of TBUFs:                 288  out of  23296     1%
Number of GCLKs:                   1  out of     16     6%
```

Figure 6.12: Rapport de synthèse de la MWST

Les ressources utilisées se situent autour de 8%, la taille maximum qu'on peut prévoir est de 75×75.

6.4 LA VÉRIFICATION FONCTIONNELLE

Actuellement, l'étape de vérification d'un circuit prend environ 70% à 80% du temps global de conception. Ce temps est souvent difficile à compresser car, contrairement à la phase de conception proprement dite, il dépend principalement du temps d'exécution des outils de vérifications. En cas d'erreur de simulation, il est très délicat de déterminer et de localiser pour un circuit de grande complexité la cause de l'erreur. Les défauts peuvent être dû par exemple, à une erreur de code, de synthèse ou de routage, car les outils CAO ne sont pas exempts de bogues.

De plus dans la majorité des cas, la vérification fonctionnelle est laissée à la charge du concepteur, le code VHDL qu'il a écrit est simulé avec un simulateur logique (Modelsim dans notre cas) en utilisant un testbench qui émule l'environnement du circuit, la vérification des chronogrammes devient de plus en plus difficile au fur et à mesure que la complexité du circuit augmente. Ce n'est pas tout, le fait que le circuit ait fonctionné correctement avec un nombre restreint de vecteurs ne prouve pas qu'il fonctionnera dans tout les cas. Il est nécessaire de faire les tests avec un nombre élevé de vecteurs afin d'assurer avec une grande probabilité la fiabilité du circuit.

Il devient alors nécessaire de développer un testbench automatique qui génère des stimulis, les applique au modèle logique du circuit, applique les mêmes stimulis au modèle mathématique et compare les résultats.

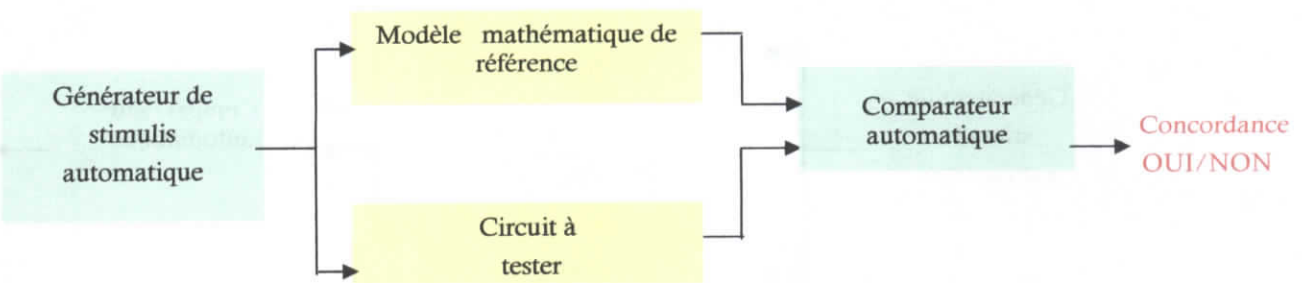


Figure 6.13 Idée de base du tesbench automatique

```
Number of Slices:                461  out of  46592     0%
Number of Slice Flip Flops:      148  out of  93184     0%
Number of 4 input LUTs:          876  out of  93184     0%
Number of bonded IOBs:            57  out of    684     8%
Number of TBUFs:                 288  out of  23296     1%
Number of GCLKs:                   1  out of     16     6%
```

Figure 6.12: Rapport de synthèse de la MWST

Les ressources utilisées se situent autour de 8%, la taille maximum qu'on peut prévoir est de 75×75.

6.4 LA VÉRIFICATION FONCTIONNELLE

Actuellement, l'étape de vérification d'un circuit prend environ 70% à 80% du temps global de conception. Ce temps est souvent difficile à compresser car, contrairement à la phase de conception proprement dite, il dépend principalement du temps d'exécution des outils de vérifications. En cas d'erreur de simulation, il est très délicat de déterminer et de localiser pour un circuit de grande complexité la cause de l'erreur. Les défauts peuvent être dû par exemple, à une erreur de code, de synthèse ou de routage, car les outils CAO ne sont pas exempts de bogues.

De plus dans la majorité des cas, la vérification fonctionnelle est laissée à la charge du concepteur, le code VHDL qu'il a écrit est simulé avec un simulateur logique (Modelsim dans notre cas) en utilisant un testbench qui émule l'environnement du circuit, la vérification des chronogrammes devient de plus en plus difficile au fur et à mesure que la complexité du circuit augmente. Ce n'est pas tout, le fait que le circuit ait fonctionné correctement avec un nombre restreint de vecteurs ne prouve pas qu'il fonctionnera dans tout les cas. Il est nécessaire de faire les tests avec un nombre élevé de vecteurs afin d'assurer avec une grande probabilité la fiabilité du circuit.

Il devient alors nécessaire de développer un testbench automatique qui génère des stimulis, les applique au modèle logique du circuit, applique les mêmes stimulis au modèle mathématique et compare les résultats.

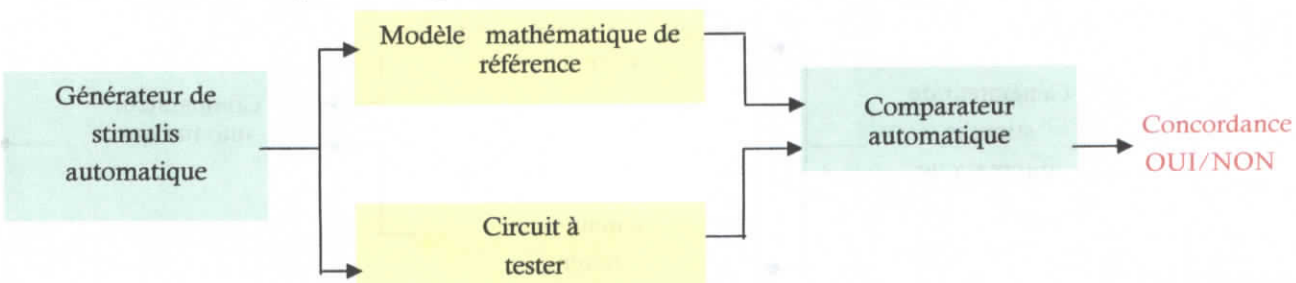


Figure 6.13 Idée de base du tesbench automatique

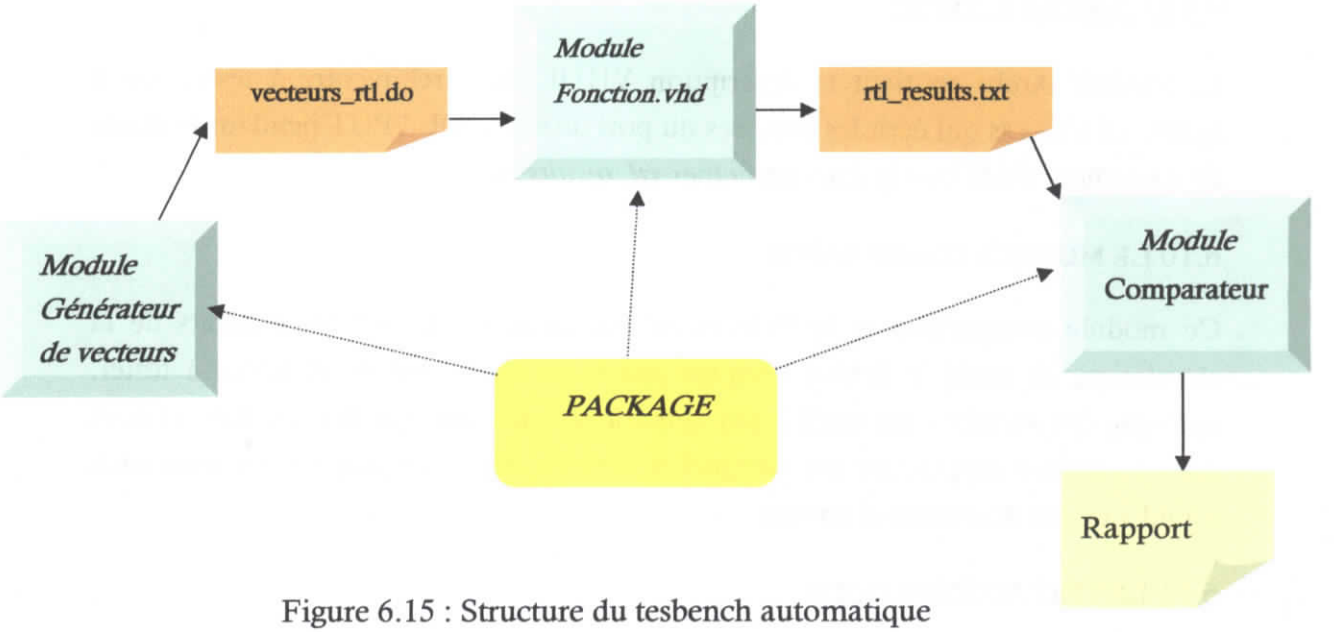


Figure 6.15 : Structure du tesbench automatique

6.7 LE PACKAGE DÉFINITIONS

Ce package contient définit les données suivantes :

- La taille de la matrice : N.
- La taille de la donnée : OpSize.
- Le nombre de vecteurs à générer pour le test : Nbre_of _Matrix.
- Le temps d'initialisation du circuit FPGA : GSR_shift (cette donnée est utilisée lors de la simulation temporelle[6]).
- Le temps de prépositionnement de la donnée : Setup.
- Les deux nombres Random1 et Random2 utilisé pour générer des échantillons aléatoires.
- Le type de donnée Matrix.
- La fonction predictor qui représente le modèle mathématique du circuit.

6.8 LE MODULE GENERATOR

Ce module a pour tâche de générer :

- Un script *vectors_rtl.do* dans un langage propre au simulateur, afin d'émuler l'environnement du circuit (signal d'horloge, signal de début, données)
- Un fichier texte *vectors_math.txt* qui contient les données à tester.

---

[6] La simulation temporelle est celle qui est faite après le placement et le routage, en incluant les délais des composants et des connexions, par opposition à la simulation fonctionnelle qui elle est faite sur le modèle comportemental qui lui contient des délais prédéfinis qui ne reflètent pas toujours la réalité.

6.9 LE MODULE ARCHI

Le module Archi contient la description VHDL de l'architecture à tester, on y ajoute un process qui écrit les données du port de sortie OUTPUT pendant la phase de traitement (SMUX=1) dans un fichier *rtl_results.txt*.

6.10 LE MODULE COMPARATOR

Ce module comparator lit le fichiers *rtl_results.txt* contenant les résultats de la simulation, lit aussi le fichier *vecteurs_maths.txt* contenant les données à tester, applique ces dernières au modèle mathématique, puis compare les résultats, et écrit dans le fichier *rapport.txt* les éventuelles erreurs, ou le message « No mismatch found » en cas d'absence d'erreurs.

6.11 LE TEST AUTOMATIQUE

Nous avons testé les différentes architectures avec ce test automatique avec 1000 matrices 6×6. Les résultats on été positifs. Nous présentons dans ce qui suit une partie des fichiers obtenus.

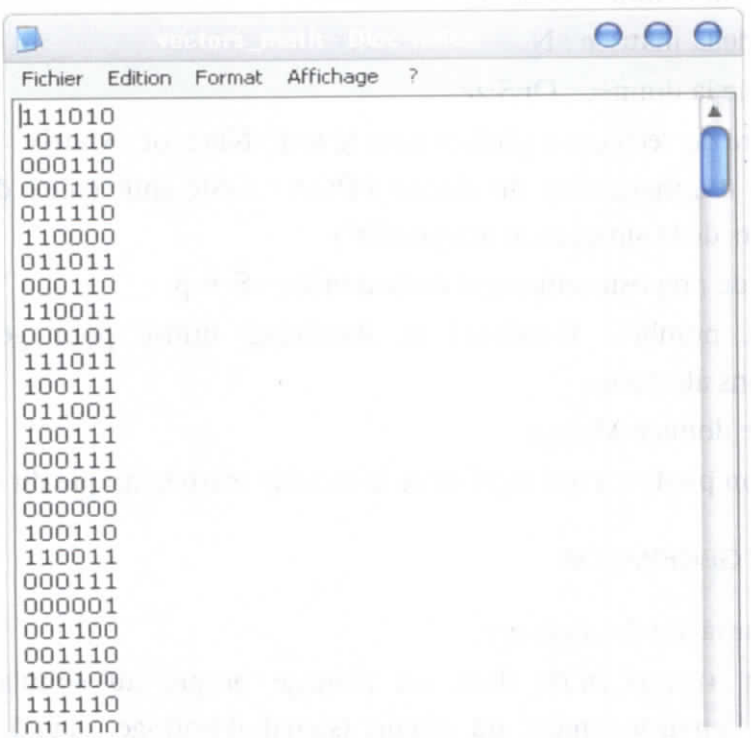


Figure 6.16 Exemple des fichiers générés par le module générateur pour une matrice 6×6 (Fermeture transitive)

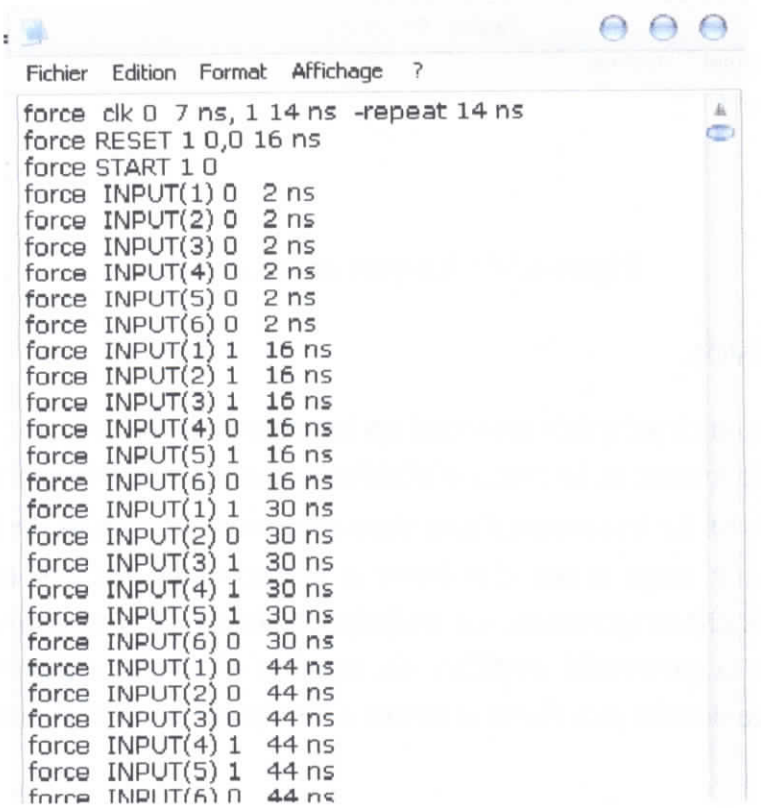


Figure 6.17 : Exemple du scripte généré par le module générateur une matrice 6×6, (Algorithme de la fermeture transitive)

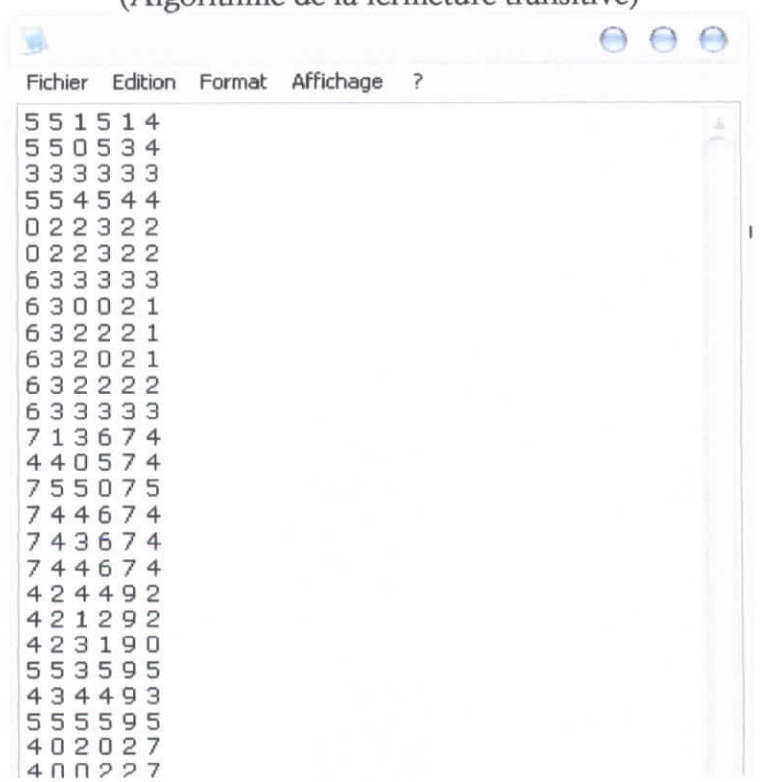


Figure 6.18 : Exemple des résultats de la simulation pour une matrice 6×6, (Algorithme de l'arbre couvrant de poids minimum)

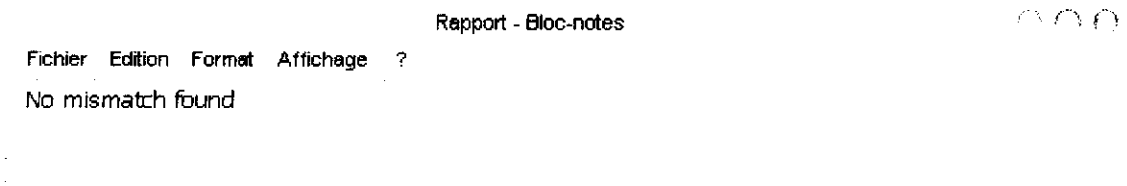


Figure 6.19 : Rapport de comparaison

6.12 CONCLUSION

Les chronogrammes présentés montrent un fonctionnement conforme au cahier des charges. Le placement et le routage n'altèrent pas ce fonctionnement (du moins pour la taille 6×6). Le test automatique vient conforter les résultats du test manuel.
Le tesbench que nous avons développé a le mérite de ne pas reposer sur la vérification des chronogrammes. La moindre erreur dans les résultats est détectée. De plus on a la possibilité d'utiliser un nombre élevé de vecteurs pour le test rendant ainsi ce dernier plus fiable et moins contraignant pour le concepteur.

CONCLUSION

Notre projet de fin d'études avait pour cahier des charges la conception d'une architecture capable d'exécuter les algorithmes du problème du chemin algébrique en N cycles d'horloge, chose qui n'avait pas été faite auparavant. Les tests ont donné des résultats concluants. On peut donc envisager d'intégrer ce circuit dans un système hôte. Ceci est d'autan plus intéressant que si on traite une application temps réel où le temps d'exécution est un paramètre primordial.

Cependant ces architectures présentent des limites du fait qu'elles soient semi-systoliques. En effet, dans une architecture systolique pure, le principe de localité est rigoureusement respecté et de ce fait, chaque cellule élémentaire est reliée avec son voisinage le plus proche seulement; ce qui n'est pas le cas dans notre architecture. Chaque cellule a une sortance de N dans le sens des lignes et N dans le sens des colonnes. De plus les commandes des buffers ont une sortance de 2N, et les commandes des multiplexeurs ont une sortance de N^2. Le temps de propagation des ces signaux de commandes risque d'engendrer des délais de propagation supérieurs au délai nécessaire pour le calcul (paramètre déterminant la fréquence maximale de fonctionnement), nous obligeant ainsi à augmenter la période. La latency ne serait donc plus linéairement proportionnelle au nombre de cellules élémentaires. On traitera dans ce cas du problème de « scalability ». De plus, dans les cas pratique les dimensions des matrices utilisées dépassent les dimensions du matériel disponible. On doit décomposer le traitement. Chaque partie peut être résolue à l'aide de l'architecture matérielle disponible. Les résultats doivent être assemblés par la suite pour donner le résultat global. C'est le « problem size independance » (ψ) sur lequel nous nous sommes malheureusement pas penché faute de temps.

RÉFÉRENCES

[1] Jong-Chuang Tsay and Pen-Yuang Chang "Some New designs of 2-D Array for Matrix Multiplication and Transitive Closure " *IEEE* Transactions on parallel and distributed systems, vol 6, NO 4, pp 351-361, April 1995.

[2] http://www.laas.fr/~lopez/cours/GRAPHES/graphes.html

[3] F.Thomson Leighton, *« Introduction to parallel algorithms and architectures »*, Morgan Kaufmann Publishers, San Mateo, California 1992.

[4] P.Quinton, Y.Robert, *« Algorithmes et architectures systoliques »*, MASSON Edition, Paris 1989.

[5] D.Rabasté *"ASIC et composants à réseaux logiques programmables, PAL, PLD, CPLD, FPGA"* IUFM d'Aix-Marseille, 2002.

[6] http://perso.wanadoo.fr/michel.hubin/physique/micro/chap.mp5.htm#fpga

[7] Documentation Xilinx

[8] Extrait du n°125 d'Electronique - Mai 2002

[9] Data sheet de la famille Virtex II

[10] A.K. Oudjida, S. Titri & K. Kaci *«Vérification Fonctionnelle»*CDTA, Avril 2004.

APPENDICE A
CODE VHDL POUR LA FERMETURE TRANSITIVE

A.1 ORGANISATION HIERARCHIQUE

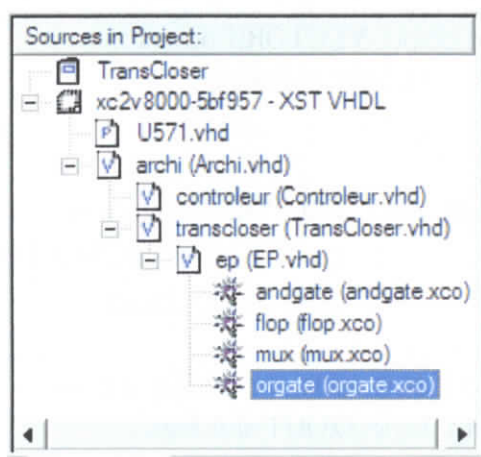


Figure A.1 : Organisation Hiérarchique pour la fermeture transitive

L'architecture globale est composée d'un séquenceur (module controleur) et d'un chemin de donnée (module transcloser). Le module transcloser est une matrice de processeurs élémentaires (module EP). Le module EP est composé d'une bascule (flop), d'une porte ET (andgate), d'une porte OU (orgate) et d'un multiplexeur (mux). Ces derniers ont été générés en utilisant le Core Generator fourni avec l'outil de développement Xilinx Foundation ISE 5.1.

A.2 CODE VHDL DU MODULE EP :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

entity EP is
   Port (Ain,L,C,CLK,S,BUFFL,BUFFC:IN  std_logic;
         Aout,OL,OC              :OUT std_logic);
end EP;
```

```
architecture Behavioral of EP is
component andgate
        port(I: IN std_logic_VECTOR(1 downto 0);
                O: OUT std_logic);
    end component;
    component orgate
        port(I: IN std_logic_VECTOR(1 downto 0);
                O: OUT std_logic);
    end component;
    component flop
        port (D: IN std_logic_VECTOR(0 downto 0);
                Q: OUT std_logic_VECTOR(0 downto 0);
                CLK: IN std_logic);
    end component;
    component BUFE -- It belongs to the library unisim
        port(E,I:IN std_logic; O:OUT std_logic);
    end component;
    component mux
        port(M: IN std_logic_VECTOR(1 downto 0);
                S: IN std_logic_VECTOR(0 downto 0);
                O: OUT std_logic);
    end component;
    signal S1,S2,S3,S4 : std_logic;
begin
AND1:andgate
    port map(I(0)=>C,I(1)=>L,O=>S1);
OR1:orgate
    port map(I(0)=>S1,I(1)=>S2,O=>S3);
M1:mux
    port map(M(0)=>S3,M(1)=>Ain,S(0)=>S,O=>S4);
D1:flop
    port map(D(0)=>S4,Q(0)=>S2,CLK=>CLK);
Buf1:BUFE
    port map(BUFFL,S2,OL);
Bufc:BUFE
    port map(BUFFC,S2,OC);
Aout <= S2;
end Behavioral;
```

Notons que les buffers trois états contenu dans les processeur élémentaires sont instantiés à partir de la librairie de primitive UNISIM.

A.3 CODE VHDL DU MODULE TRANSCLOSER

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use WORK.U571.all;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
--library UNISIM;
--use UNISIM.VComponents.all;

entity TransCloser is
        Port (INPUT :IN  std_logic_vector(1 to N);
                                        OUTPUT:OUT std_logic_vector(1 to N);
                                        SBUFF :IN  std_logic_vector(1 to N);
                                        CLOCK :IN  std_logic;
                                        SMUX  :IN  std_logic);
end TransCloser;

architecture Behavioral of TransCloser is
        component EP
        Port (Ain,L,C,CLK,S,BUFFL,BUFFC:IN  std_logic;
                        Aout,OL,OC              :OUT std_logic);
        end component;
        signal LIG,ROW :std_logic_vector(1 to N);
        type TwoDAr is array(1 to N,1 to N+1)of std_logic;
        signal LOAD : TwoDAr;
begin
        L: for I in 1 to N generate
                C: for J in 1 to N generate
                        EP1:EP

        port map(LOAD(I,J),LIG(I),ROW(J),CLOCK,SMUX,SBUFF(J),SBUFF(I),LOAD(I,J+1
),LIG(I),ROW(J));
                end generate;
        end generate;
        ES:
                for I in 1 to N generate
                        LOAD(I,1)<=INPUT(I);
                        OUTPUT(I)<=LOAD(I,N+1);
```

```
        end generate;
end Behavioral;
```

A.4 CODE VHDL DU MODULE CONTROLEUR

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use Work.U571.ALL;
-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

entity Controleur is
   Port (CLK,RESET : IN std_logic;
                                    SMUX : OUT std_logic;
                                    SBUFF: OUT std_logic_VECTOR(0 to N-1));
end Controleur;

architecture Behavioral of Controleur is
        type StateType is(LOAD,TREAT);
        signal State,NextState:StateType;
        signal Count:integer range 0 to N-1;
begin
        process (CLK,RESET)
        begin
                if (RESET='1') then
                        State <= LOAD;
                        elsif (CLK'event and CLK='1') then
                        State <= NextState;
                        end if;
        end process;

        process(CLK,RESET)
        begin
                if (CLK'event and CLK='1') then
                        if (RESET='1') then
                                Count <=0;
                        elsif (Count/=N-1) then
                                Count <= Count+1;
```

```
                else
                        Count <=0;
                end if;
        end if;
end process;

process (State,Count)
begin
        if (State=LOAD) then
                if(Count=N-1)then
                        Nextstate <= TREAT;
                else
                        NextState <= LOAD;
                end if;
        elsif (State=TREAT) then
                if (Count=N-1)then
                        NextState <= LOAD;
                else
                        NextState <= TREAT;
                end if;
end if;
end process;

process (State)
begin
        if (State=LOAD) then
                SMUX <= '1';
        elsif (State=TREAT) then
                SMUX <= '0';
        end if;
end process;

process (State,Count)
begin
        if (State=TREAT) then
                for j in 0 to N-1 loop
                        if (j=Count) then
                        SBUFF(j) <='1';
                        else
                        SBUFF(j) <='0';
                        end if;
                end loop;
```

```
            elsif (State=LOAD) then
                SBUFF <= (others =>'0');
            end if;
        end process;
end Behavioral;
```

A. 5 CODE VHDL DU MODULE ARCHI

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;
use WORK.U571.all;

entity Archi is
        Port (INPUT :IN  std_logic_vector(1 to N);
                                OUTPUT:OUT std_logic_vector(1 to N);
                                CLK  :IN std_logic;
                                RESET:IN std_logic);
end Archi;

architecture Behavioral of Archi is
        component controleur
            Port (CLK  :IN std_logic;
                --START:IN std_logic;
                                RESET:IN std_logic;
                                SMUX:OUT std_logic;
                                SBUFF:OUT std_logic_VECTOR(1 to N ));
        end component;
        component transcloser
            Port (INPUT :IN  std_logic_vector(1 to N);
                                OUTPUT:OUT std_logic_vector(1 to N);
                                SBUFF :IN  std_logic_vector(1 to N);
                                CLOCK :IN  std_logic;
                                SMUX  :IN  std_logic);
        end component;
signal S1:std_logic;
```

```
signal S2:std_logic_VECTOR(1 to N);
begin
U1:controleur
        port map (CLK,RESET,S1,S2);
U2:transcloser
        port map (INPUT,OUTPUT,S2,CLK,S1);
end Behavioral;
```

A.6 CODE VHDL DU PACKAGE U571

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package U571 is
-- Declare constants
 constant N: integer :=6;
end U571;
package body U571 is
end U571;
```

Ce package sert à modifier la taille de la matrice sans passer par chaque module.

APPENDICE B
CODE VHDL POUR LA LONGUEUR DU CHEMIN LE PLUS COURT

B.1 ORGANISATION HIÉRARCHIQUE

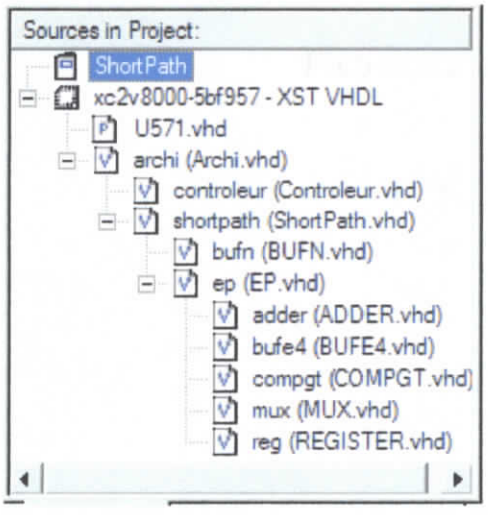


Figure B.1 : Organisation Hiérarchique pour la longueur du chemin le plus court

L'architecture globale est composée d'un séquenceur (module controleur) et d'un chemin de donnée (module shortpath). Le module shortpath est une matrice de processeurs élémentaires (module EP). Le module EP est composé d'un registre (register), d'un additionneur (adder), d'un comparateur (compgt), de multiplexeurs (mux), et de buffers 3 états à M bits -M taille de la donnée- (bufe4).

B.2 CODE VHDL DU MODULE EP

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use WORK.U571.ALL;
-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

entity EP is
    Port (Ain,L,C   :IN  std_logic_VECTOR(M-1 downto 0);
-- INPUT port used recpectively to propagate date during the loading,
-- and per row & column during the computation
```

```
                Aout,OL,OC:OUT std_logic_VECTOR(M-1 downto 0);
-- OUTPUT port used recpectively to propagate date during the loading,
-- and per row & column during the computation
                                CLK,BUFFL,BUFFC:IN std_logic;
-- Clock & enbale of 3state buffer L(ligne),C(Colonne)
                                S :IN std_logic);
-- Select MUX.
end EP;

architecture Behavioral of EP is
        component ADDER          --4 bit adder
                port( A: IN std_logic_VECTOR(M-1 downto 0);
        B: IN std_logic_VECTOR(M-1 downto 0);
                        C_OUT: OUT std_logic;
                        S: OUT std_logic_VECTOR(M-1 downto 0));
         -- S = Sum, C_OUT = Carry out
        end component;
        component COMPGT -- comparator (A_GT_B=1 when A>B)
                port( A: IN std_logic_VECTOR(M-1 downto 0);
                  B: IN std_logic_VECTOR(M-1 downto 0);
                  A_GT_B: OUT std_logic);
        end component;
        component REG -- 4 bit register
                port( D: IN std_logic_VECTOR(M-1 downto 0);
                        Q: OUT std_logic_VECTOR(M-1 downto 0);
                        CLK: IN std_logic);
        end component;
        component BUFE4    -- 4 bit 3state buffer
                port( O:OUT std_logic_VECTOR(M-1 downto 0);
                                E:IN std_logic;
                                I:IN std_logic_VECTOR(M-1 downto 0) );
        end component;
        component MUX -- 4 bit 2 to 1 multiplexer
                port( MA: IN std_logic_VECTOR(M-1 downto 0);
                        MB: IN std_logic_VECTOR(M-1 downto 0);
                        S: IN std_logic;
                        O: OUT std_logic_VECTOR(M-1 downto 0));
        end component;
        component AND2B1 -- and gate with the port I0 inverted
                port( O: OUT std_logic;
                                I0: IN std_logic;
                                I1: IN std_logic);
```

```
        end component;
        signal S1,S2,S3,S4 : std_logic_VECTOR(M-1 downto 0);
        signal S5,S6,S7 : std_logic;
begin
ADD:ADDER
        port map(A     => L,
                                        B     => C,
                                        C_OUT => S5,
                                        S     => S1);
CMP:COMPGT
        -- this comparator, the AND1, & MO are used to perform the min function
        port map(A      => S2,
                 B      => S1,
                                        A_GT_B => S6);
AND1:and2b1
        port map(O  => S7,
                                        I0 => S5,
                                        I1 => S6);
M0:MUX
        port map(MA => S2,
                                        MB => S1,
                                        S  => S7,
                                        O  => S3);
M1:MUX
        port map(MA => S3,
                                        MB => Ain,
                                        S  => S,
                                        O  => S4);
D1:REG
        port map(D   => S4,
                                        Q   => S2,
                                        CLK => CLK);
Buf1:BUFE4
        port map(O => OL,
                                        E =>   BUFFL,
                                        I => S2);
Bufc:BUFE4
        port map(O => OC,
                                        E => BUFFC,
                                        I => S2);
Aout <= S2;
end Behavioral;
```

B.3 CODE VHDL DU MODULE SHORTPATH

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use WORK.U571.ALL;

-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

entity ShortPath is
   Port (INPUT :IN  BUS4;
                              -- Port from which we introduce the matrix to be treated
                              OUTPUT:OUT BUS4;
                              -- Port from which we can see the results
                              SBUFF :IN  std_logic_VECTOR(1 to N);
                              -- Slect 3state buffer, in order to propagate
                              -- data per row and per column
                              CLOCK :IN  std_logic;
                              SMUX  :IN  std_logic);
                              -- Select Multiplexer
end ShortPath;

architecture Behavioral of ShortPath is
component EP
        Port (Ain,L,C :IN  std_logic_VECTOR(M-1 downto 0);
                CLK : IN std_logic;
                              S :IN std_logic;
                              BUFFL,BUFFC : IN std_logic;
                              Aout,OL,OC:OUT std_logic_VECTOR(M-1 downto 0));
        end component;
        component BUFN
                Port (I : IN std_logic_VECTOR(1 to N);
                              O :OUT std_logic_VECTOR(1 to N));
        end component;
        component BUFG
                Port (I : IN std_logic;
                              O :OUT std_logic);
        end component;
```

```
        component BUF
                Port (I : IN std_logic;
                                O :OUT std_logic);
        end component;
        signal SBUFF1: std_logic_VECTOR(1 to N);
        signal CLK,SMUX1:std_logic;
--type BUS4 is array (1 to N)of std_logic_VECTOR(0 to 3);
        signal LIG,ROW :BUS41;
        -- signals used to link the EP's, used to propagate
        -- data during computation
        type TwoDAr is array(1 to N,1 to N+1)of std_logic_VECTOR(M-1 downto 0);
        signal LOAD : TwoDAr;
        -- signal used to link the EP's, used to propagate the
        -- data during the loading
        signal SO: std_logic;

begin
        CLKBUF:BUFG
        port map(CLOCK,CLK);
        BUFF1:BUFN
        port map(SBUFF,SBUFF1);
        BUFF2:BUF
        port map(SMUX,SMUX1);
        L: for I in 1 to N generate
                C: for J in 1 to N generate
                                EP1:EP

        port map(LOAD(I,J),LIG(I),ROW(J),CLK,SMUX1,SBUFF1(J),SBUFF(I),LOAD(I,J+1),
LIG(I),ROW(J));
                end generate;
        end generate;

        ES:for I in 1 to N generate
                        LOAD(I,1) <= CONV_STD_LOGIC_VECTOR(INPUT(I),M);
                        OUTPUT(I) <= CONV_INTEGER(LOAD(I,N+1));
                end generate;
end Behavioral;
```

Notons l'utilisation de buffers afin d'amplifier les signaux de commande.

Les modules controleurs et archi sont les même que pour la fermeture transitive, sauf que le type de la donnée change dans le module archi

B.4 CODE VHDL DU PACKAGE U571

```
package U571 is
-- Declare constants
  constant N: integer := 6; -- taille de la matrice.
  constant M: integer := 4; -- M taille de la donnÈe
-- Declare user types
  type BUS4 is array (1 to N)of integer range 0 to 2**M-1 ;
        type BUS41 is array (1 to N) of std_logic_VECTOR (M-1 downto 0);
end U571;
```

APPENDICE C
CODE VHDL POUR L'ARBRE COUVRANT DE POIDS MINIMUM

C.1 ORGANISATION HIERARCHIQUE

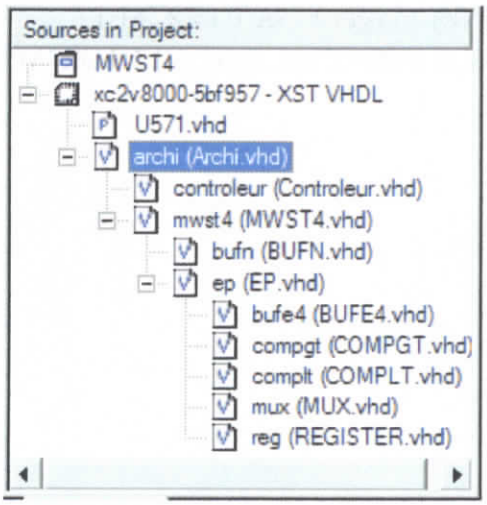


Figure C.1 : Organisation hiérachique

L'architecture globale est composée d'un séquenceur (module controleur) et d'un chemin de donnée (module mwst4). Le module mwst4 est une matrice de processeurs élémentaires (module EP). Le module EP est composé d'un registre (register), de comparateurs (compgt et complt), de multiplexeurs (mux), et de buffers 3 états à M bits -M taille de la donnée- (bufe4).

C.2 CODE VHDL DU MODULE EP

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.u571.all;
-- Uncomment the following lines to use the declarations that are
-- provided for instantiating Xilinx primitive components.
library UNISIM;
use UNISIM.VComponents.all;

entity EP is
    Port (Ain,L,C   :IN  std_logic_VECTOR(M-1 downto 0);
-- INPUT port used recpectively to propagate date during the loading,
-- and per row & column during the computation
```

```
                Aout,OL,OC:OUT std_logic_VECTOR(M-1 downto 0);
-- OUTPUT port used recpectively to propagate date during the loading,
-- and per row & column during the computation
                                CLK,BUFFL,BUFFC:IN std_logic;
-- Clock & enbale of 3state buffer L(ligne),C(Colonne)
                                S :IN std_logic);
-- Select MUX.
end EP;

architecture Behavioral of EP is
        component complt      -- comparator (A_LT_B=1 when A<B)
                port( A: IN std_logic_VECTOR(3 downto 0);
                  B: IN std_logic_VECTOR(3 downto 0);
                  A_LT_B: OUT std_logic);
        end component;
        component compgt -- comparator (A_GT_B=1 when A>B)
   port( A: IN std_logic_VECTOR(3 downto 0);
                        B: IN std_logic_VECTOR(3 downto 0);
                        A_GT_B: OUT std_logic);
  end component;
        component reg -- 4 bit register
                port( D: IN std_logic_VECTOR(M-1 downto 0);
                        Q: OUT std_logic_VECTOR(M-1 downto 0);
                        CLK: IN std_logic);
        end component;
        component BUFE4    -- 4 bit 3state buffer
                port( O:OUT std_logic_VECTOR(3 downto 0);
                                        E:IN std_logic;
                                        I:IN std_logic_VECTOR(3 downto 0) );
        end component;
        component mux -- 4 bit 2 to 1 Multiplexer
                port( MA: IN std_logic_VECTOR(3 downto 0);
                        MB: IN std_logic_VECTOR(3 downto 0);
                        S: IN std_logic;
                        O: OUT std_logic_VECTOR(3 downto 0));
        end component;
        signal S1,S2,S3,S4 : std_logic_VECTOR(0 to 3);
        signal S5,S6 : std_logic;
begin
CMP1:complt
-- we use a this comparator and a mux in order to perform the max function.
        port map(A => L,
```

```
                                        B => C,
                                        A_LT_B => S5);
CMP2:compgt
-- we use a this comparator and a mux in order to perform the min function.
        port map(A =>S2,
                B => S1,
                                A_GT_B => S6);
M0:mux
        port map(MA => S2,
                                        MB => S1,
                                        S => S6,
                                        O => S3);
M1:mux
        port map(MA => S3,
                MB => Ain,
                                S => S,
                                O => S4);
M2:mux
        port map(MA =>L,
                                        MB =>C,
                                        S => S5,
                                        O => S1);
D1:reg
        port map(D => S4,
                                        Q => S2,
                                        CLK => CLK);
Buf1:BUFE4
        port map(OL,BUFFL,S2);
Bufc:BUFE4
        port map(OC,BUFFC,S2);
Aout <= S2;
end Behavioral;
```

Le reste des modules ont les mêmes codes que précedement.

APPENDICE D
CODE VHDL POUR LE TESTBENCH AUTOMATIQUE

D.1 CODE VHDL DU PACKAGE DEFINITIONS

Exemple pour le test de l'arbre couvrant de poids minimum

```
library IEEE;
use STD.TEXTIO.ALL;                                    -- DÈfini les primitives
 d'Écriture et de lecture
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.MATH_REAL.ALL;

package Definitions is                                  -- contient toutes les i
nformations d'implÈmentation

  constant N : integer:=6;                              -- taille de la matrice
  constant Nbr_of_Matrix: integer:=100;                 -- nombre de matrices a
tester
  constant OpSize : integer:=4;
  constant Period      : time := 25 ns;
  constant Latency: integer :=N;                        -- temps de rÈponse

  constant GSR_shift: time :=200 ns;                    -- temps d'initialisation
 du circuit FPGA
  constant Setup : time :=10 ns;                        -- temps de prÈ-positio
nnement de la donnÈe

  constant Random1 : integer := 150;                    -- permet de sÈlection
ner les vecteur de faÁon alÈatoire
  constant Random2 : integer := 3777;

  type Matrix is array (1 to N, 1 to N) of integer range 0 to 2**OpSize-1;

  function Predictor (A:IN Matrix) return Matrix;       -- reprÈsente les donnÈes de
 la fonction mathÈmatique ‡ implÈmenter
  function Min (A,B:IN integer) return integer;
```

```
  function Max(A,B:IN integer)return integer;
end Definitions;

package body Definitions is

  function Min(A,B: integer) return integer is
  variable C : integer;
  begin
        if ( A<B) then
              C:=A;
        else
              C:=B;
        end if;
        return C;
  end Min;

  function Max(A,B: integer) return integer is
  variable C : integer;
  begin
        if ( A>B) then
              C:=A;
        else
              C:=B;
        end if;
        return C;
  end Max;

  function Predictor (A: Matrix) return Matrix is
  variable C,D : Matrix;
  begin
        for i in 1 to N loop
              for j in 1 to N loop
                    C(i,j):=A(i,N-j+1);
              end loop;
        end loop;

        for k in 1 to N loop
              for i in 1 to N loop
                    for j in 1 to N loop
                          D(i,j):=Min(C(i,j),Max(C(i,k) , C(k,j)));
                    end loop;
              end loop;
```

```
                C:=D;
        end loop;

    for i in 1 to N loop
                for j in 1 to N loop

        C(i,j):=D(i,N-j+1);          -- reprÈsente la fonction mathÈmatique a implÈmenter
                end loop;
        end loop;
        return C;
  end Predictor;
end Definitions;
```

D.2 CODE VHDL DU MODULE GENERATOR

```
library IEEE;
use STD.TEXTIO.ALL;                  -- DÈfini les primitives d'Ècriture et de lecture
use IEEE.MATH_REAL.ALL;
use Work.Definitions.all;                    -- Fait appel au package definition

entity generator is
end generator;

architecture Behavioral of generator is

  file vectors_rtl: text open write_mode is "C:/Xilinx/bin/MWST/vectors_rtl.do";        -
-CrÈation du fichier d'Ècriture du modËle RTL
  file vectors_math: text open write_mode is "C:/Xilinx/bin/TestFonctionnel/vectors_m
ath.txt";  -- Creation du  fichier d'Ècriture du modËle mathÈmatique




begin
 gen: process
   --  variable temps: time := Setup;                                          -- active
r pour la simulation fonctionnelle
     variable temps: time := GSR_shift+Setup;                                  -- activ
er pour la simulation temporelle
     variable simtime: time;                                                   -- temps
 de simulation
     variable remaining_Matrix: integer;              -- nombre de matrices
```

```
variable seed1 : integer:=random1;
variable seed2 : integer:=random2;
variable INPUT : integer range 0 to 2**(OpSize)-1;
--  declaration des variables RTL
variable random: real;
variable buff_rtl, buff_math: line;
variable j: integer;

begin                -- Permet de dÈfinir  le chronogramme de l'horloge
 write(buff_rtl,string'("force  clk 0  "));
 write(buff_rtl,Period/2);
 write(buff_rtl,string'(", 1 "));
 write(buff_rtl,Period);
 write(buff_rtl,string'("  -repeat "));
 write(buff_rtl,Period);
 writeline(vectors_rtl, buff_rtl);
 write(buff_rtl,string'("force RESET 1 0"));
 write(buff_rtl,string'(",0 "));
 write(buff_rtl,Period+GSR_shift+Setup);
 writeline(vectors_rtl, buff_rtl);
 --write(buff_rtl,string'("force START 1 0"));
 --writeline(vectors_rtl, buff_rtl);
 for I in 1 to N loop
    write(buff_rtl,string'("force  INPUT("));
    write(buff_rtl,I);
    write(buff_rtl,string'(") "));
    write(buff_rtl,0);
    write(buff_rtl,string'("    "));
    write(buff_rtl,temps);
    writeline(vectors_rtl, buff_rtl);
 end loop;
 temps:=temps+Period;

    -- Permet de gÈnÈrer alÈatoirement les valeurs de la donnÈe "a" mathÈmatique
et RTL
 remaining_Matrix:=Nbr_of_Matrix;
 while remaining_Matrix > 0 loop
 for J in 1 to N loop
        for I in 1 to N loop
            INPUT:=0;
            for L in  0 to Opsize-1 loop
                uniform(seed1,seed2, random);
```

```
                if (0.5<=random) then
                        INPUT:=INPUT+2**L;
                        end if;
                    end loop;
                write(buff_rtl,string'("force  INPUT("));
                write(buff_rtl,I);
                write(buff_rtl,string'(") "));
                write(buff_rtl,INPUT);
                write(buff_rtl,string'("    "));
                write(buff_rtl,temps);
                        writeline(vectors_rtl, buff_rtl);
            write(buff_math,INPUT);
                                write(buff_math,string'(" "));
        end loop;
    writeline(vectors_math, buff_math);
    if (J/=N) then
        temps:=temps+Period;
    elsif(J=N) then
        temps:=temps+(N+1)*Period;
    end if;
    end loop;
    remaining_Matrix:=remaining_Matrix-1;
    end loop;
     simtime:= temps+latency*Period;
     write(buff_rtl,string'("    "));
     writeline(vectors_rtl, buff_rtl);
     write(buff_rtl,string'("run "));
     write(buff_rtl,simtime);
     writeline(vectors_rtl,buff_rtl);
    wait;
    end process;
 end behavioral;
```

D.2 CODE VHDL DU MODULE COMPARATOR

```
library IEEE;
                                                -- DÈfini les primitives d'Écriture et de l
ecture
use IEEE.MATH_REAL.ALL;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.ALL;
use STD.TEXTIO.ALL;
use work.Definitions.all;                                    -- Fait appel au
package dÈfinition

entity comparator is
end comparator;

architecture Behavioral of comparator is

--file rtl_result : text open read_mode is "C:/Xilinx/bin/TestFonctionnel/rtl_results.txt";
                -- CrÈation du fichier de lecture du model RTL
file rtl_result : text open read_mode is "C:/Xilinx/bin/TestFonctionnel/rtl_results_ba.txt
";
file logfile: text open write_mode    is "C:/Xilinx/bin/TestFonctionnel/Rapport.txt";
        -- CrÈation du fichier d'Écriture contenant les erreurs
file math_data: text open read_mode    is "C:/Xilinx/bin/TestFonctionnel/vectors_math.
txt";

begin

 process
 variable buff_rtl, buff_math, buff_log: line;
 variable buff_rtl1: Matrix ;
 variable buff_math1 : Matrix;
 variable tmp:bit_vector(1 to N);
 variable tmp1:std_logic_VECTOR (1 to N);
 variable tmp2:std_logic_VECTOR (1 to N);
 variable a_math: Matrix;                                    --declar
ation des variable mathÈmatique
  variable mismatch : boolean :=false;
 variable temps: time :=Setup+(2*N)*Period;                  --pour la
 simulation fonctionnelle
 --variable temps: time :=Setup+(2*N)*Period+GSR_shift;       --
pour la simulation temporelle
 variable I,J,k: integer;
 variable random: real;
 variable seed1 : integer:=random1;
 variable seed2 : integer:=random2;

 begin
```

```
while not(endfile(rtl_result)) loop
for I in 1 to N loop
        readline(rtl_result,buff_rtl);              -- lecture des rÈsulats RTL
        for J in 1 to N loop
                read(buff_rtl,buff_rtl1(I,J));
        end loop;
end loop;

for I in 1 to N loop

        readline(math_data,buff_math);          -- lecture de la matrice ‡ passer pour la fo
nction mathÈmatique
                for J in 1 to N loop
                        read(buff_math,a_math(I,J));
                end loop;
end loop;
buff_math1:=predictor(a_math);              -- Calcul des rÈsultats mathÈmatiques
--
for I in 1 to N loop
        for J in 1 to N loop

        if abs(buff_math1(i,j) - buff_rtl1(i,j) ) /= 0  --Comparaison entre les rÈsultats RT
L et mathÈmatique
                then mismatch:=true;
                write(buff_log,string'("Mismatch at Time:  "));
                write(buff_log, temps);
                writeline(logfile, buff_log);
                write(buff_log, string'(" "));
                writeline(logfile, buff_log);
                write(buff_log,string'("RTL value is :  "));
                write(buff_log,buff_rtl1(I,J));
                writeline(logfile, buff_log);
                write(buff_log,string'("Math value is:  "));
                write(buff_log,buff_math1(I,J));
                writeline(logfile, buff_log);
                write(buff_log, string'(" "));
                writeline(logfile, buff_log);

        write(buff_log, string'("_____________________________________________________
_____________________ "));
                writeline(logfile, buff_log);
                write(buff_log, string'(" "));
```

```
                writeline(logfile, buff_log);
                end if;
        end loop;
 end loop;
 temps:=temps+period;
end loop;
if not(mismatch) then write(buff_log,string'("No Mismatch Found"));
                writeline(logfile, buff_log);
end if;
wait;
end process;
end Behavioral;
```

Pour plus de comodités, nous avons choisi d'enrober le module archi dans un module testbench

D.3 CODE VDHL DU MODULE TESTBENCH

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use STD.TEXTIO.ALL;
use work.U571.ALL;
library SIMPRIM;
use SIMPRIM.VCOMPONENTS.ALL;
use SIMPRIM.VPACKAGE.ALL;
use work.definitions.ALL;

entity testbench is
        generic (N:integer :=6);
        Port (  INPUT :IN  BUS4;

        --OUTPUT:INOUT BUS4;                                          -- activ
er pour la simulation fonctionnelle

        OUTPUT:INOUT STD_LOGIC_VECTOR2 ( 6 downto 1 , 3 downto 0 );      -- ac
tiver pour la simulation temporelle
                CLK  :IN STD_LOGIC;
                SMUX :INOUT STD_LOGIC;
                SBUFF:OUT STD_LOGIC_VECTOR(1 to 6);
                RESET:IN STD_LOGIC);
```

```
                RESET:IN STD_LOGIC);
end testbench;

architecture test of testbench is

-- file rtl_result:text open write_mode is "C:/Xilinx/bin/TestFonctionnel/rtl_results.txt"
;
file rtl_result:text open write_mode is "C:/Xilinx/bin/TestFonctionnel/rtl_results_ba.tx
t";
component archi
        --Port (INPUT :IN  BUS4;                    -- activer pour la simulation fonction
nelle
        --                  OUTPUT:OUT BUS4;
        --                  CLK  :IN std_logic;
        --                  SMUX:OUT std_logic;
        --                  SBUFF:OUT std_logic_VECTOR(1 to N);
                    --      RESET:IN std_logic);
        Port (
        smux : out STD_LOGIC;           -- activer pour la simulation temporelle
                reset : in STD_LOGIC ;
                clk : in STD_LOGIC ;
                sbuff : out STD_LOGIC_VECTOR ( 1 to 6 );
                output : out STD_LOGIC_VECTOR2 ( 6 downto 1 , 3 downto 0 );
                input : in STD_LOGIC_VECTOR2 ( 6 downto 1 , 3 downto 0 ) );
        end component;
signal TMP1:STD_LOGIC_VECTOR2 ( 6 downto 1 , 3 downto 0 ); -- activer pour la si
mulation temporelle
-- signal TMP2:integer range 0 to 2**M-1;
begin
L : for I in 1 to N generate
        C: for J in 0 to M-1 generate

        TMP1(I,J) <= CONV_STD_LOGIC_VECTOR(INPUT(I),M)(J);  -- activer pour l
a simulation temporelle
        end generate;
end generate;

A:archi
        Port map    (
        input  => TMP1,-- INPUT pour la simulation fonctionnelle
                        output =>OUTPUT,
                        clk => CLK,
```

```
                    );

process (CLK)
variable buff_rtl:line;
variable TMP:integer range 0 to 2**M-1;
begin
        if (CLK'event and CLK='1')then
                if ( SMUX='1') then
                        if (now>(2*N+1)*period+Setup+GSR_shift) then
                                for I in 1 to N loop
                                        TMP:=0;
                                        for J in 0 to M-1 loop

        TMP:=TMP+CONV_INTEGER(OUTPUT(I,J))*(2**J);  -- activer pour la simulatio
n temporelle
                                        end loop;
                                        write(buff_rtl,TMP);
                                        write(buff_rtl,string'(" "));
                                end loop;
                                writeline(rtl_result,buff_rtl);

--                              for I in 1 to N loop
--
        write(buff_rtl,OUTPUT(I));        -- activer pour la simulation fonctionnelle
--                                      write(buff_rtl,string'(" "));
--                              end loop;
--                              writeline(rtl_result,buff_rtl);
                        end if;
                else
                        null;
                end if;
        end if;

        end process;
end test;
```