



ENSTA

IP PARIS

INSTITUT POLYTECHNIQUE DE PARIS  
ENSTA PARIS

---

# CSC\_5RO07\_TA, Multiprocessors on Chip

Vivado 2019.1 - Final Project

---

by

Gianluca BAGHINO, Daniel FRULANE,  
Natalia GALLEGOS, Diego PINCER et Guilherme TROFINO

supervised by  
Omar HAMMAMI  
Hervé LE PROVOST

**Confidentiality Notice**  
Non-confidential and publishable report

ROBOTIQUE  
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET COMMUNICATION

---

Paris, FR  
6 janvier 2025

# Table des matières

---

<b>1 Q1 : Network on Chip 3x3</b>	<b>2</b>
1.1 Mise en oeuvre et validation du NOC 3x3 avec générateurs de trafic . . . . .	2
1.2 Analyse de la latence et de la bande passant . . . . .	3
1.3 NOC sur Zynq . . . . .	4
1.4 Teste et validation du NOC sur la zedboard. . . . .	7
<b>2 Q2 : Multicoeur 3 coeurs : 1 ARM et 2 MicroBlaze</b>	<b>8</b>
2.1 Traitement d'image : filtre Sobel . . . . .	8
2.2 Flux de travail . . . . .	8
2.3 Diagramme de blocs . . . . .	10
2.4 Mémoires . . . . .	11
2.5 Codes . . . . .	12
2.5.1 Structure et Organisation du Système . . . . .	12
2.5.2 Collaboration entre l'ARM et les MicroBlaze . . . . .	13
2.5.3 Gestion de la Mémoire BRAM et des Sémaphores . . . . .	13
2.5.4 Traitement de l'Image . . . . .	13
2.6 Résultats . . . . .	14
2.6.1 Avant l'application du filtre Sobel . . . . .	14
2.6.2 Après l'application du filtre Sobel . . . . .	14
2.6.3 Ce que l'on doit attendre comme résultat . . . . .	14
2.6.4 Temps de traitement . . . . .	15
2.7 Système multi-horloge . . . . .	15
<b>3 Q3 : Multicoeur 5 coeurs : 1 ARM et 4 MicroBlaze</b>	<b>18</b>
3.1 Flux de travail . . . . .	18
3.2 Diagramme de blocs . . . . .	19
3.2.1 Processeurs MicroBlaze . . . . .	19
3.2.2 Mémoire Locale pour les Processeurs MicroBlaze . . . . .	19
3.2.3 BRAM et Contrôleurs . . . . .	19
3.2.4 AXI Interconnects . . . . .	20
3.2.5 Débogage des MicroBlaze et Réinitialisation du Système . . . . .	20
3.2.6 Vue d'Ensemble du Système . . . . .	20
3.3 Mémoires . . . . .	20
3.4 Résultats . . . . .	21
3.4.1 Temps de traitement . . . . .	21
<b>4 Q4 : Exploration Automatisée</b>	<b>22</b>
4.1 Data Generation . . . . .	22
4.1.1 Software Optimizations . . . . .	22
4.2 Data Analysis . . . . .	23
4.2.1 Database Creation . . . . .	23
4.3 Résultats Q2 . . . . .	24
4.3.1 Duration . . . . .	24
4.3.2 Power . . . . .	24
4.3.3 WNS . . . . .	25
4.3.4 Ressources . . . . .	26
4.4 Résultats Q3 . . . . .	27
4.4.1 Duration . . . . .	27
4.4.2 Power . . . . .	27
4.4.3 WNS . . . . .	28
4.4.4 Ressources . . . . .	29
4.5 Résultats Généraux . . . . .	29

## 1. Q1 : Network on Chip 3x3

Le Network on Chip (NoC) est un concept clé dans les systèmes multiprocesseurs sur puce (MPSoC), abordé dans le cadre de la matière "Multiprocesseurs sur puce". Il s'agit d'une solution d'interconnexion utilisant un réseau de routeurs et de liens pour assurer une communication efficace entre les différents coeurs de traitement et autres modules intégrés sur une même puce. Pendant les travaux pratiques avec la ZedBoard, cette technologie est étudiée pour comprendre comment les topologies de NoC optimisent les performances, l'évolutivité et l'efficacité énergétique dans des systèmes embarqués complexes.

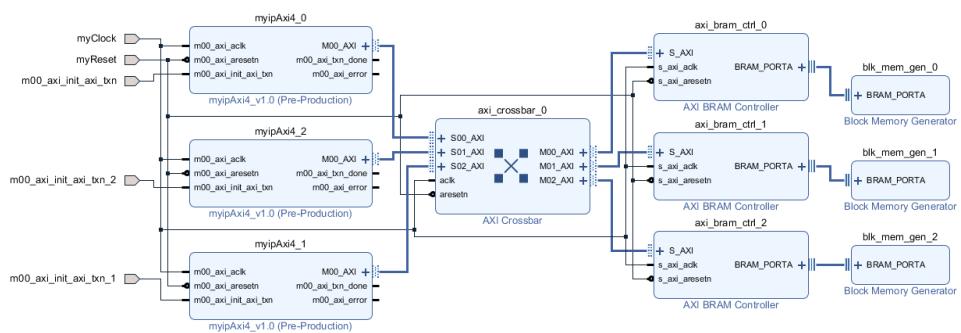


FIGURE 1.1 : Implémentation Full Crossbar.

Le projet est appelé NoC 3x3 en raison de son architecture suivant une topologie en grille 3x3, où trois éléments principaux (comme les modules "myip\_v1.0") sont interconnectés à trois contrôleurs de mémoire BRAM via un AXI Crossbar. Cette configuration reflète l'organisation typique d'un Network on Chip, où chaque noeud du réseau peut communiquer efficacement avec les autres, garantissant une haute scalabilité et performance. Cette topologie a été implémentée et validée dans le projet à l'aide de la ZedBoard.

### 1.1. Mise en œuvre et validation du NOC 3x3 avec générateurs de trafic

Pour tester et valider le NoC 3x3, l'exemple disponible sur le drive du cours a été téléchargé à partir du dossier "crossBar", qui contient trois sous-dossiers supplémentaires. Pour cette étape, ainsi que pour le calcul de la latence et de la bande passante, le modèle "crossbarSimple" a été utilisé, ce dernier n'étant pas encore relié à la carte. Le fichier ouvert pour les simulations se trouve à l'emplacement suivant : "/crossBar/crossbarSimple/axiCrossBar/axiCrossBar.xpr". Pendant le processus, il a été nécessaire de permettre à l'application de mettre à jour certains IPs pour qu'ils soient compatibles avec la version actuelle. Enfin, la simulation (behavioral simulation) a été exécutée pour observer les résultats de trafic, lesquels peuvent également être visualisés à partir de l'onglet "Flow".



FIGURE 1.2 : Écran de la “Behavior Simulation”.

## 1.2. Analyse de la latence et de la bande passante

L'analyse de la latence et de la bande passante du NoC 3x3 prend en compte le concept de "slack", qui correspond à la différence de temps entre le début du prochain cycle d'horloge et le début de la réponse au signal. Si le slack est positif, la réponse intervient avant le deuxième cycle d'horloge ; s'il est négatif, elle intervient après. La latence, quant à elle, est définie comme l'intervalle entre l'entrée d'un signal dans le générateur de trafic (TX) et son écriture dans la mémoire BRAM, ce qui indique que les données ont été correctement enregistrées. Cette latence est influencée par le temps de programmation du système. Pour l'analyse, un burst de 256 mots de 32 bits a été considéré, avec une fréquence d'horloge de 100 MHz (10 ns). Les résultats de la simulation, observés sur l'écran de la Behavioral Simulation, montrent que la latence mesurée est d'environ 80 ns.

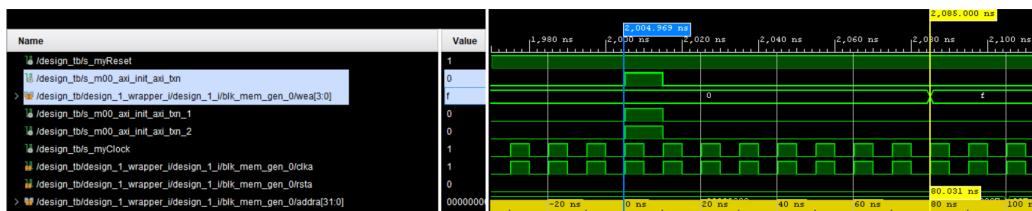


FIGURE 1.3 : Mésurement de la latence sur l'écran “Behavior Simulation”.

La bande passante est la capacité maximale de transmission de données dans un système pendant un intervalle de temps donné, généralement mesurée en bits par seconde (bps). Dans le contexte du NoC, elle indique le volume de données pouvant être transféré entre les composants. Elle représente la capacité du canal de communication et est directement liée à l'efficacité du système.

$$\text{Bande passante} = \frac{\text{Volume total de données (bits)}}{\text{Temps total de transfert (secondes)}}$$

- Volume total de données :** Chaque transfert de données comprend 256 mots, et chaque mot contient 32 bits. Ainsi, le volume total de données est donné par :

$$\text{Volume total de données} = 256 \times 32 = 8192 \text{ bits.}$$

- Temps total de transfert :** Chaque transfert de mot prend 10 ns. Le temps total pour transférer 256 mots est donc :

$$\text{Temps total de transfert} = 256 \times 10 \text{ ns} = 2560 \text{ ns} = 2.56 \times 10^{-6} \text{ s.}$$

En appliquant ces valeurs à la formule, nous obtenons :

$$\text{Bande passante} = \frac{8192}{2,56 \times 10^{-6}} = 3.2 \text{ Gbps.}$$

Ainsi, la bande passante mesurée pour ce système est de **3 x 3.2 Gbps = 9.6 Gbps**.

### 1.3. NOC sur Zynq

La figure suivante présente l'implementation du circuit NOC sur Zynq.

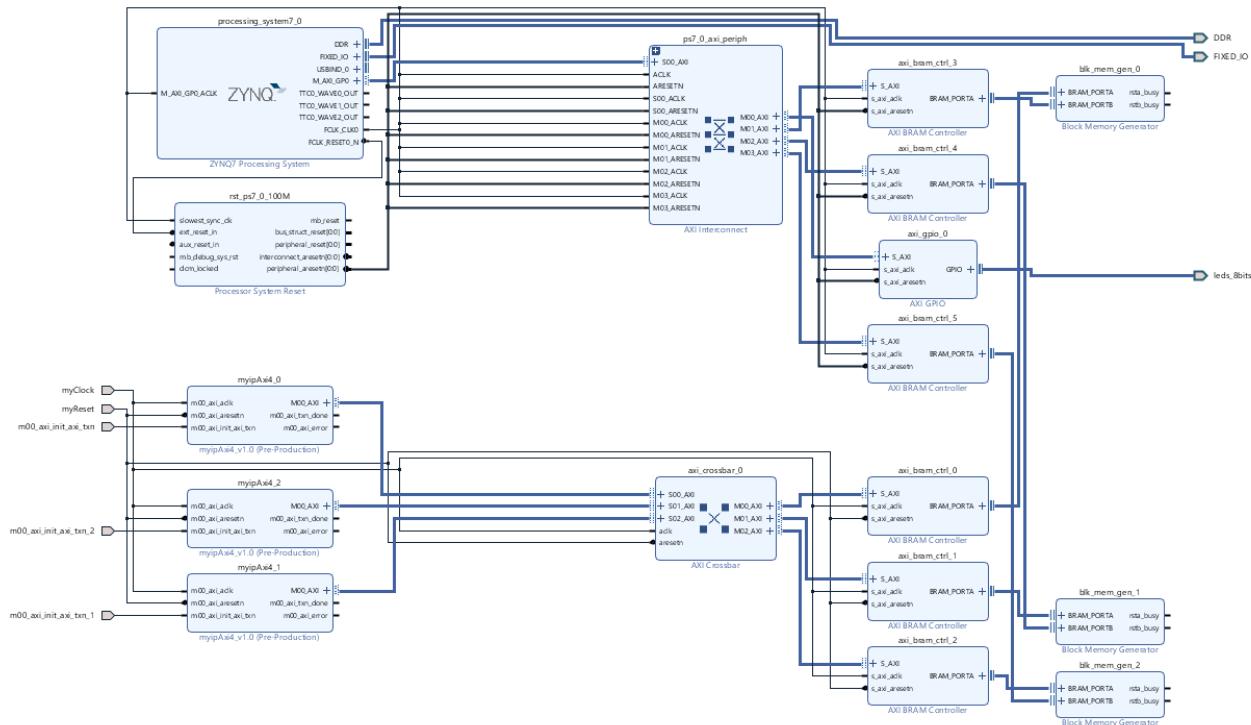


FIGURE 1.4 : NOC sur ZYNQ

Lié aux blocs de mémoire BRAM, nous trouvons les blocs de gestion de mémoire, ceux qu'iront s'en charger de synchroniser les opérations d'accès mémoire, sachant que les deux parties du circuit peuvent opérer à différents fréquences d'horloge. En plus, l'AXI Interconnect est responsable pour le traitement des données sortantes du Processeur Zynq et les mémoires et GPIO.

Les configurations de Crossbar Full-Access et Shared Access ont été testées pour vérifier leur utilisation de ressources. La stratégie Synthesis Performance Net Delay Low a été choisi pour vérifier si le chemin de connexions dans le circuit pourrait changer pour un nouveau critère de synthèse :

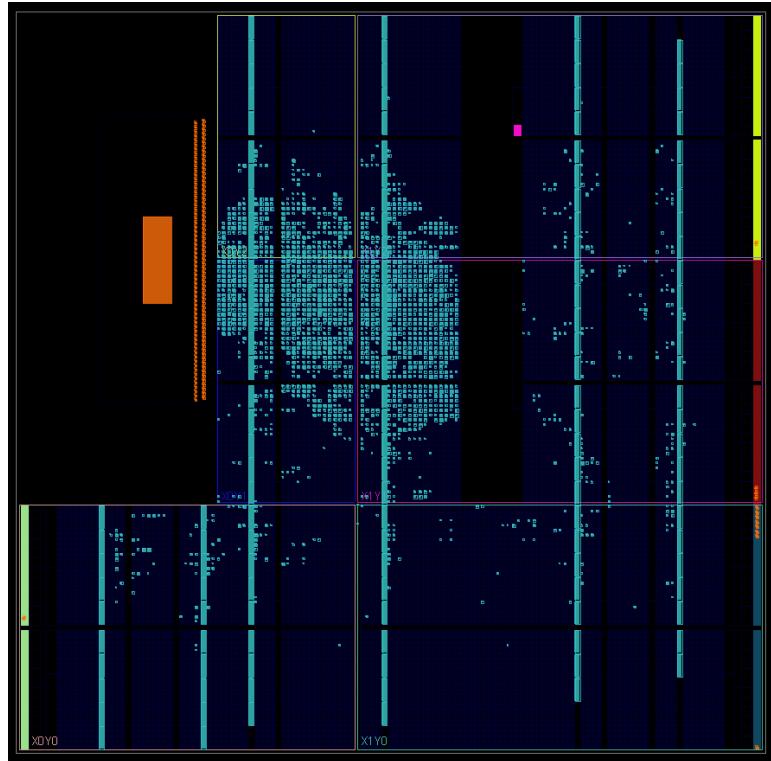


FIGURE 1.5 : Implémentation Full-Access Crossbar.

Resource	Utilization	Available	Utilization %
LUT	4689	53200	8.81
LUTRAM	243	17400	1.40
FF	4273	106400	4.02
BRAM	132	140	94.29
IO	13	200	6.50

FIGURE 1.6 : Utilisation des ressources pour la configuration Full-Access crossbar.

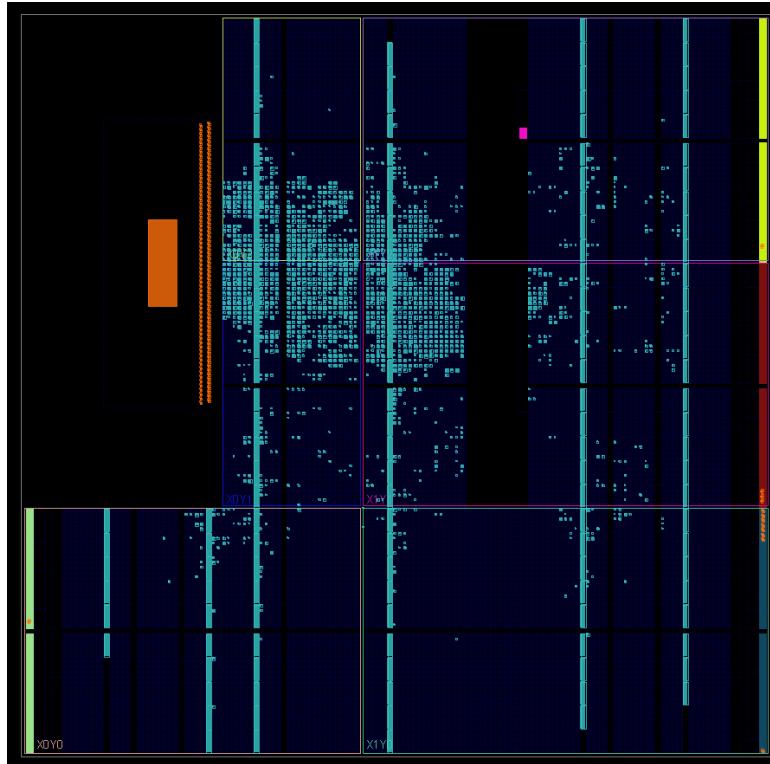


FIGURE 1.7 : Implémentation Shared-Acess Crossbar.

Resource	Utilization	Available	Utilization %
LUT	3973	53200	7.47
LUTRAM	223	17400	1.28
FF	3930	106400	3.69
BRAM	132	140	94.29
IO	13	200	6.50

FIGURE 1.8 : Utilisation des ressources pour la configuration Shared-Acess Crossbar.

Vu que les ressources BRAM et IO restent égales à cause de la fixation de l'architecture en termes de connexions et mémoire, la légère différence entre ressources (surtout les LUT) ne doit pas justifier le changement de configuration Crossbar, sachant que la Shared-Access est bien moins performant.

Finalement, les stratégies de synthèse ont faire varier plus considérablement les ressources utilisés pour la configuration Full-Access Crossbar. Même si la proportion générale entre ressources a resté similaire. L'optimisation de Area peut bien réduire, en termes proportionnels, l'utilisation des LUT, LUTRAM et FF ; néanmoins, cette option a comme effet une réduction de performance :

Resource	Utilization	Available	Utilization %
LUT	3221	53200	6.05
LUTRAM	111	17400	0.64
FF	3346	106400	3.14
BRAM	132	140	94.29
IO	13	200	6.50

FIGURE 1.9 : Synthesis Area Optimize High.

Resource	Utilization	Available	Utilization %
LUT	4219	53200	7.93
LUTRAM	243	17400	1.40
FF	4210	106400	3.96
BRAM	132	140	94.29
IO	13	200	6.50

FIGURE 1.10 : Synthesis Performance Optimize High.

## 1.4. Teste et validation du NOC sur la zedboard.

Pour implémenter et valider la solution, un flux standard a été suivi dans Vivado : tout d'abord, l'implémentation a été exécutée (Run Implementation) et le bitstream a été généré (Generate Bitstream). Ensuite, le matériel a été exporté en utilisant la fonction Export Hardware, incluant le fichier bitstream. En exécutant le SDK, un nouvel Application Program en langage C a été créé et configuré pour communiquer avec le système. Dans le menu Xilinx, l'option Program FPGA a été utilisée pour charger le bitstream sur la ZedBoard. Enfin, le programme a été exécuté directement sur le hardware en utilisant l'option Run As ↳ Launch on Hardware. L'objectif était de permettre au code de lire les valeurs stockées dans les mémoires BRAM, configurées au préalable dans l'Address Editor.

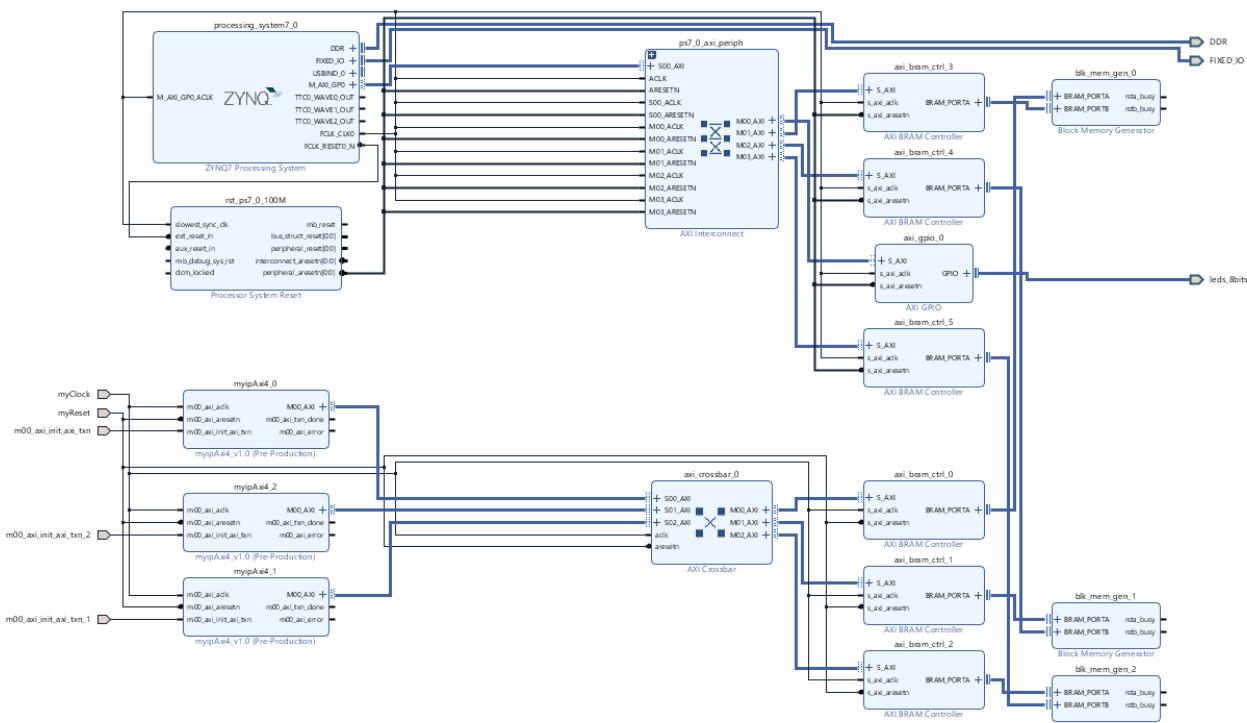


FIGURE 1.11 : NOC sur ZYNQ

## 2. Q2 : Multicoeur 3 coeurs : 1 ARM et 2 MicroBlaze

### 2.1. Traitement d'image : filtre Sobel

Pour tester le système multicoeur, il a été décidé de l'utiliser dans une tâche de traitement d'image, car en raison de sa nature, où il est nécessaire de faire plusieurs multiplications tout au long du processus, cette option semble correcte pour tester autant que possible les possibilités et les possibilités portée du système.

Sachant cela, il a été décidé d'utiliser le filtre Sobel pour détecter les bords de l'image ; Ceci est basé sur l'identification de changements brusques d'intensité lumineuse dans une image. Cette méthode utilise des masques (ou noyaux) qui permettent de calculer les gradients d'intensité dans deux directions principales : horizontale ( $G_x$ ) et verticale ( $G_y$ ) (Figure 2.1).

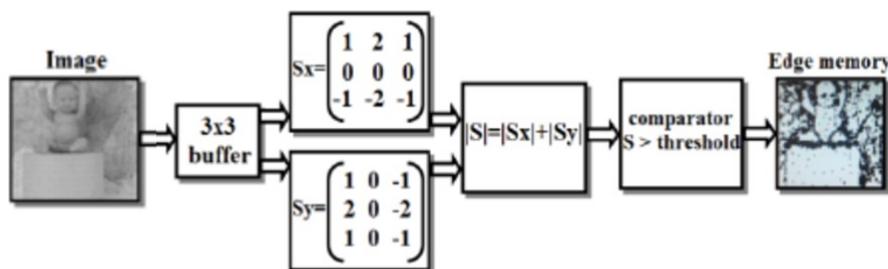


FIGURE 2.1 : Schéma explicatif du flux de travail du système.

Pour appliquer le filtre Sobel, l'image originale doit être en niveaux de gris, puisque les calculs de dégradé sont effectués sur des valeurs d'intensité. Les masques de filtre sont de petites matrices qui mettent en évidence le taux de changement de l'intensité des pixels dans chaque direction. Le masque horizontal ( $G_x$ ) est utilisé pour détecter les bords verticaux, tandis que le masque vertical ( $G_y$ ) met en évidence les bords horizontaux. Ces masques convoluent avec l'image originale, ce qui signifie qu'ils se déplacent pixel par pixel à travers l'image, multipliant leurs valeurs par celles des pixels correspondants du voisinage, puis additionnant les résultats. Il en résulte une image qui met en évidence des zones de changement brusque d'intensité dans la direction correspondante à chaque masque.

Après avoir appliqué les deux masques, les résultats  $G_x$  et  $G_y$  sont combinés pour calculer l'ampleur du dégradé total à chaque pixel. L'ampleur du gradient, donnée par la formule  $G = \sqrt{G_x^2 + G_y^2}$ , représente l'intensité des bords. Plus la valeur de pente est élevée, plus le bord est raide à cet endroit. Cela génère une nouvelle image en niveaux de gris dans laquelle les bords sont clairement visibles.

Le filtre Sobel est populaire en raison de sa simplicité et de son efficacité. Ce filtre est un outil essentiel en vision par ordinateur, utilisé dans des tâches telles que la détection de contours, la détection d'objets et l'analyse d'images médicales, entre autres.

### 2.2. Flux de travail

Le flux de traitement est expliqué ci-dessous, qui implique la transmission, le partitionnement, le traitement parallèle et la reconstruction d'une image grâce à la collaboration entre un ordinateur, un ARM 9 et deux processeurs MicroBlaze. Chaque étape du processus est détaillée ci-dessous et est visible dans la figure 2.2 :

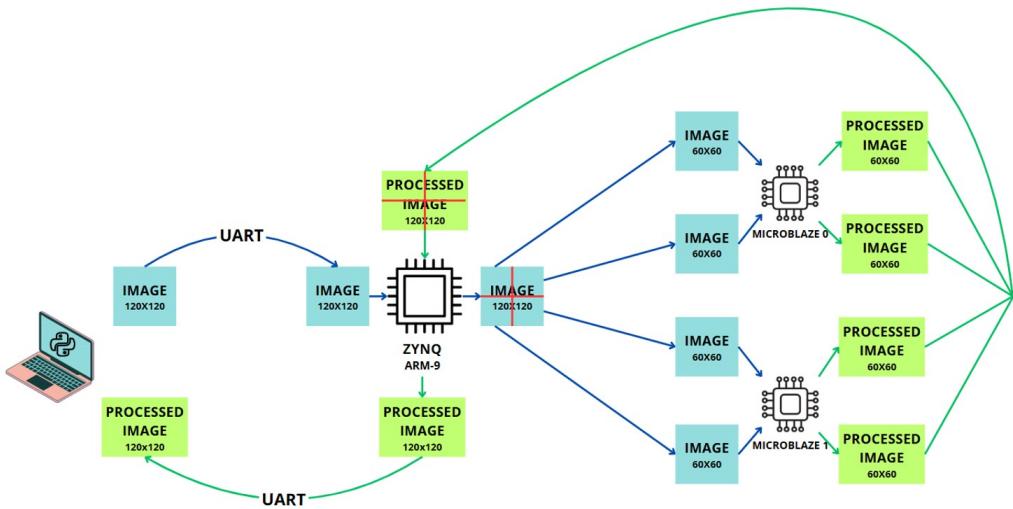


FIGURE 2.2 : Schéma explicatif du flux de travail du système.

1. L'ordinateur fait office de source de données initiale, envoyant une image à l'ARM 9 via l'interface UART.
2. Dès réception de l'image complète, l'ARM 9 la divise en quatre sous-images de dimensions égales. Cette partition est destinée à répartir la charge de traitement uniformément entre les processeurs MicroBlaze disponibles. Chaque sous-image est mappée sur une région spécifique de mémoire partagée entre l'ARM 9 et le MicroBlaze.
3. L'ARM 9 attribue deux sous-images à chaque processeur MicroBlaze :
  - Le \*\*MicroBlaze 0\*\* reçoit deux sous-images correspondant à la moitié supérieure de l'image (les deux premiers quarts).
  - Le \*\*MicroBlaze 1\*\* s'occupe des deux sous-images restantes, correspondant à la moitié inférieure de l'image (les deux derniers quarts). Cette répartition égale permet aux deux processeurs de travailler simultanément sur des tâches de traitement spécifiques.
4. Chaque MicroBlaze effectue des opérations pour utiliser le filtre Sobel sur les sous-images attribuées. Ces opérations consistent en plusieurs multiplications matricielles, additions et calculs de racine. Le résultat de ce traitement est deux sous-images traitées par chaque MicroBlaze.
5. Une fois le traitement terminé, les MicroBlazes écrivent les sous-images traitées dans les régions désignées de la mémoire partagée. Cette mémoire fait office de moyen d'échange de données entre les processeurs et l'ARM 9.
6. L'ARM 9 récupère les sous-images traitées de la mémoire partagée et les combine pour reconstruire l'image complète de 120 x 120 pixels. Cette étape garantit que les quatre sous-images sont réassemblées à leur emplacement d'origine pour former une représentation cohérente de l'image traitée.
7. Enfin, l'ARM 9 transmet l'image traitée à l'ordinateur via l'interface UART.

## 2.3. Diagramme de blocs

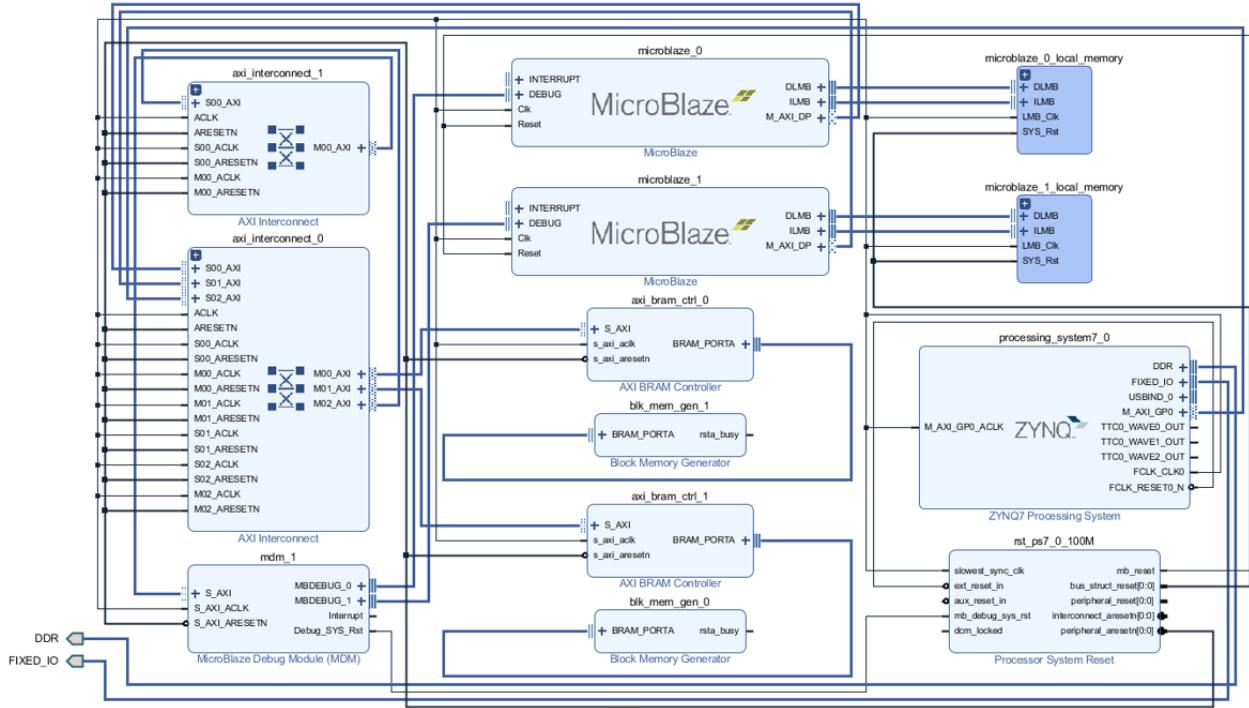


FIGURE 2.3 : Diagramme de blocs du système multicoeur 1 ARM et 2 Microblaze

Le système présenté combine un processeur **ZYNQ7 Processing System** et deux processeurs **MicroBlaze**, conçu pour le traitement d'images. Le **ZYNQ7**, avec ses deux coeurs ARM Cortex-A9, agit comme le noyau principal, gérant les tâches générales, la communication avec les périphériques et le transfert de données vers les mémoires DDR et les blocs logiques programmables. Sa puissance en fait un choix idéal pour manipuler de grands volumes de données, comme les images, tout en coordonnant les autres blocs.

Les processeurs **MicroBlaze** apportent de la flexibilité au système et sont spécialement configurés pour effectuer la détection des contours dans les images. Ils fonctionnent en parallèle, se partageant les calculs des matrices nécessaires au traitement. Cette approche parallèle optimise les performances et accélère considérablement les opérations. La capacité des MicroBlaze à s'adapter à des tâches spécifiques et à opérer de manière autonome en fait des éléments essentiels de ce design.

Pour relier tous les composants, les **AXI Interconnects** agissent comme des autoroutes, organisant le trafic des données entre les processeurs, les mémoires et les périphériques. Cela est crucial, car le traitement d'images exige des transferts rapides et efficaces de grandes quantités de données. De plus, le **MicroBlaze Debug Module** permet de vérifier que les processeurs MicroBlaze fonctionnent correctement, facilitant le débogage pendant le développement et garantissant que les tâches de traitement spécifiques sont réalisées sans erreur.

Le module **Processor System Reset** garantit que le système démarre de manière ordonnée et synchronisée, évitant tout dysfonctionnement au démarrage et assurant la stabilité globale du design. Chaque MicroBlaze dispose de sa propre mémoire locale (*DLMB* et *ILMB*), ce qui leur permet de fonctionner de façon indépendante et rapide, en stockant les données et les instructions nécessaires au traitement en temps réel.

Par ailleurs, les **BRAM (Block RAM)**, gérées par les **AXI BRAM Controllers** et les **Block Memory Generators**, jouent un rôle fondamental en tant que buffers rapides. Elles permettent de stocker temporairement des fragments d'images ou des résultats intermédiaires pendant que les MicroBlaze effectuent les calculs nécessaires à la détection des contours, réduisant la latence et augmentant l'efficacité du système.

Dans l'ensemble, ce design exploite la puissance du ZYNQ pour les tâches générales, la capacité des Mi-

croBlaze pour un traitement dédié, les BRAM pour un stockage rapide, et les interconnects pour un flux de données ordonné. C'est une solution robuste et évolutive, idéale pour le traitement d'images, où la rapidité et la capacité à gérer de grands volumes de données sont essentielles.

## 2.4. Mémoires

Le système implémente une structure de mémoire hiérarchique conçue pour optimiser l'interaction entre deux processeurs MicroBlaze et un système ZYNQ basé sur un ARM Cortex-A9. Ce schéma organise et segmente la mémoire pour diviser efficacement la charge de travail entre différents éléments, facilitant ainsi le traitement parallèle et la gestion des ressources.

Le système utilise deux principaux types de mémoire : Chaque MicroBlaze a accès à deux BRAM locales :

- DLMB (Data Local Memory Bus) : Pour stocker et accéder aux données.
- ILMB (Instruction Local Memory Bus) : Exclusif pour les instructions du programme.

Ces mémoires sont conçues pour un accès rapide et sont directement associées à chaque MicroBlaze, améliorant ainsi l'efficacité des opérations locales.

DDR (Dynamic RAM) : la mémoire DDR est gérée par l'ARM 9 et est utilisée pour stocker les images, à la fois en entrée et en sortie, et pour la coordination entre les éléments du système. A partir de cette mémoire, sont définis deux blocs qui serviront au transfert de données entre l'ARM 9 et les deux Microblaze. Cette application est réalisée en parallélisant complètement l'accès à la mémoire du Microblaze, ainsi les données utilisées par le premier Microblaze se trouvent dans un bloc mémoire spécifique (**0x40000000 - 0x40007fff**) et les données du deuxième Microblaze dans un autre bloc (**0x42000000 - 0x42007fff**). Cela garantit que pendant le traitement de l'image, il n'y a pas de conflits d'entrée en mémoire, améliorant ainsi le temps d'exécution et évitant les plantages. Pour cette raison, il a été défini que les masques de convolution utilisés dans le traitement, ( $G_x$ ) et ( $G_y$ ), sont stockés de manière redondante dans les BRAM de chaque MicroBlaze.

Toutes les mémoires locales (DLMB et ILMB) et les blocs DDR ont été configurés avec une taille de 32 Ko. Cette valeur a été sélectionnée en fonction de la taille des images à traiter, garantissant suffisamment d'espace pour stocker les données nécessaires sans gaspiller de ressources. Ces valeurs sont visibles sur la figure 2.4.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 1G)					
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	32K	0x4000_7FFF
axi_bram_ctrl_1	S_AXI	Mem0	0x4200_0000	32K	0x4200_7FFF
mdm_1	S_AXI	Reg	0x4340_0000	4K	0x4340_0FFF
microblaze_0					
Data (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	32K	0x4000_7FFF
axi_bram_ctrl_1	S_AXI	Mem0	0x4200_0000	32K	0x4200_7FFF
microblaze_0_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	32K	0x0000_7FFF
mdm_1	S_AXI	Reg	0x4340_0000	4K	0x4340_0FFF
Instruction (32 address bits : 4G)					
microblaze_0_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	32K	0x0000_7FFF
microblaze_1					
Data (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	32K	0x4000_7FFF
axi_bram_ctrl_1	S_AXI	Mem0	0x4200_0000	32K	0x4200_7FFF
microblaze_1_local_memory/dlmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	32K	0x0000_7FFF
mdm_1	S_AXI	Reg	0x4340_0000	4K	0x4340_0FFF
Instruction (32 address bits : 4G)					
microblaze_1_local_memory/ilmb_bram_if_cntlr	SLMB	Mem	0x0000_0000	32K	0x0000_7FFF

FIGURE 2.4 : Répartition de la mémoire.

Le flux de traitement de la mémoire est :

1. Les images d'entrée sont initialement stockées dans des régions spécifiques du DDR.
2. L'ARM 9 distribue les sous-images aux BRAM correspondantes du MicroBlaze.
3. Chaque MicroBlaze traite les sous-images attribuées en utilisant ses mémoires locales (DLMB et ILMB) et les masques ( $G_x$ ) et ( $G_y$ ).
4. Les résultats du traitement sont stockés dans des régions de sortie du DDR, d'où l'ARM 9 les récupère pour reconstruire l'image complète.

Cette structure hiérarchique permet un traitement efficace en tirant pleinement parti des capacités de parallélisme du système. La segmentation de la mémoire garantit une gestion ordonnée des ressources, tandis que la redondance des données dans les BRAM élimine les conflits d'accès, réduisant ainsi le temps d'exécution et améliorant les performances globales du système.

## 2.5. Codes

Le système est composé de trois parties principales : le processeur ARM et deux processeurs MicroBlaze, chacun jouant un rôle essentiel dans le traitement parallèle de l'image. Cette architecture distribuée permet d'effectuer des tâches complexes de traitement d'image de manière efficace et coordonnée, en tirant parti de la parallélisation et de la synchronisation.

### 2.5.1. Structure et Organisation du Système

Le processeur ARM agit comme le contrôleur principal du système. Il est responsable de la réception de l'image depuis l'hôte (un ordinateur ou une source externe), de la division de l'image en quatre parties égales et de la gestion de la communication et de la synchronisation entre les processeurs MicroBlaze. En termes d'organisation, l'ARM envoie les différentes parties de l'image aux mémoires locales des MicroBlaze et veille

à ce que les noyaux Sobel (pour détecter les bords dans l'image) soient envoyés aux MicroBlaze pour leur traitement. De plus, l'ARM gère l'utilisation des sémaphores (flags), qui sont essentiels pour la synchronisation entre les MicroBlaze et l'ARM.

De leur côté, chaque processeur MicroBlaze est chargé de traiter une section spécifique de l'image (un "quart" de l'image complète). Chaque MicroBlaze a accès à une fraction de l'image stockée dans la mémoire BRAM et applique le filtre Sobel sur cette partie. Ce processus se fait de manière indépendante et parallèle, ce qui permet au système de traiter toute l'image rapidement.

### 2.5.2. Collaboration entre l'ARM et les MicroBlaze

La collaboration entre l'ARM et les MicroBlaze repose sur une communication via des sémaphores, qui sont des variables de contrôle utilisées pour gérer l'accès à la mémoire partagée et synchroniser les tâches entre les processeurs. L'ARM utilise ces sémaphores pour indiquer aux MicroBlaze quand commencer à traiter une partie de l'image et aussi pour s'assurer que chaque MicroBlaze termine son travail avant de passer à l'étape suivante. Ce processus de synchronisation est crucial pour éviter les conflits d'accès à la mémoire et garantir que les données ne soient pas écrasées ou corrompues.

Lorsque l'ARM reçoit l'image de l'hôte, il la divise en quatre parties et les distribue dans la mémoire BRAM, attribuant chaque section à un des MicroBlaze. Les sémaphores sont utilisés pour coordonner l'exécution entre les MicroBlaze, en indiquant quand ils doivent commencer à traiter. Par exemple, l'ARM envoie un signal à un MicroBlaze pour commencer à traiter la première section de l'image, et une fois que ce dernier a terminé, l'ARM peut envoyer un signal au MicroBlaze suivant pour qu'il traite la section suivante, et ainsi de suite.

Une fois que les MicroBlaze ont terminé le traitement de leurs sections respectives de l'image, ils mettent à jour leur sémaphore pour indiquer à l'ARM qu'ils ont fini. L'ARM, de son côté, attend que tous les MicroBlaze aient terminé leur travail avant de reconstruire les résultats et de les renvoyer à l'hôte.

### 2.5.3. Gestion de la Mémoire BRAM et des Sémaphores

La mémoire BRAM (Block RAM) est utilisée pour stocker à la fois l'image d'origine et les résultats traités des différentes sections de l'image. Chaque section de l'image (un quart de l'image complète) est stockée dans différentes zones de la mémoire BRAM, ce qui permet aux MicroBlaze de travailler de manière indépendante sur leur propre section sans interférer les uns avec les autres.

Les sémaphores sont essentiels pour la gestion correcte de cette mémoire. Chaque MicroBlaze dispose d'un sémaphore de contrôle qui indique son état (qu'il soit en attente, en traitement ou terminé). Par exemple, l'ARM vérifie l'état de ces sémaphores pour savoir quand les MicroBlaze ont terminé de traiter leurs sections et quand il peut commencer à reconstruire l'image traitée. Les sémaphores aident également à éviter que les MicroBlaze n'accèdent simultanément aux mêmes zones de la mémoire, ce qui garantirait l'intégrité des données.

En plus de la synchronisation entre l'ARM et les MicroBlaze, les sémaphores permettent à chaque MicroBlaze de contrôler le début et la fin de son traitement. Lorsque l'ARM envoie un signal à un MicroBlaze pour commencer à traiter une section de l'image, le sémaphore correspondant est activé. Une fois que le MicroBlaze termine le traitement, il change le sémaphore pour indiquer qu'il a fini sa tâche, permettant ainsi à l'ARM ou au MicroBlaze suivant de commencer son travail.

### 2.5.4. Traitement de l'Image

Le traitement de l'image se déroule en deux grandes étapes : la réception des données et l'application du filtre Sobel. Lors de la première étape, l'ARM reçoit l'image de l'hôte et la divise en quatre sections égales, que chaque MicroBlaze va traiter individuellement. Ces sections sont envoyées dans la mémoire BRAM locale de chaque MicroBlaze, où elles seront traitées.

Dans la deuxième étape, chaque MicroBlaze applique le filtre Sobel sur sa section respective. Le filtre Sobel est un opérateur de convolution qui permet de détecter les bords d'une image en calculant les gradients d'intensité dans deux directions : la direction horizontale (X) et la direction verticale (Y). Le filtre Sobel utilise

deux noyaux de convolution distincts. Le premier noyau est utilisé pour calculer le gradient dans la direction horizontale (X), et le second noyau est utilisé pour calculer le gradient dans la direction verticale (Y). Ces noyaux sont généralement les suivants :

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \quad G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

En appliquant ces noyaux à chaque pixel de la section d'image, on obtient deux valeurs : le gradient horizontal  $G_x$  et le gradient vertical  $G_y$ . La magnitude du bord au niveau de chaque pixel est ensuite calculée en combinant ces deux gradients à l'aide de la formule suivante :

$$M = \sqrt{G_x^2 + G_y^2}$$

Cela donne la magnitude du bord, qui indique l'intensité du changement de couleur à cet endroit. Un gradient élevé correspond généralement à une zone où il y a un changement significatif, comme un bord ou une ligne, tandis qu'un gradient faible correspond à une zone plus homogène.

Ce processus est appliqué indépendamment par chaque MicroBlaze à sa propre section de l'image, permettant ainsi un traitement parallèle efficace. Chaque processeur MicroBlaze obtient donc une version "filtrée" de sa section de l'image, où les bords sont clairement définis. Une fois le traitement terminé, chaque MicroBlaze met à jour son sémaphore pour indiquer qu'il a fini. L'ARM récupère ensuite les résultats de chaque MicroBlaze, les combine pour reconstruire l'image complète et l'envoie à l'hôte.

Cette organisation parallèle permet un traitement plus rapide de l'image, car chaque MicroBlaze traite une section distincte de manière indépendante. La gestion des sémaphores assure que chaque étape se déroule dans le bon ordre, garantissant ainsi l'intégrité des données et une synchronisation efficace entre l'ARM et les MicroBlaze.

## 2.6. Résultats

### 2.6.1. Avant l'application du filtre Sobel

Avant l'application du filtre Sobel, l'image originale est intacte, telle qu'elle a été reçue depuis l'hôte. À ce stade, l'image contient tous les détails de la scène ou de l'objet, montrant des zones de couleurs continues, comme des cieux dégagés ou des surfaces uniformes, ainsi que des bords bien définis entre différents objets ou régions de l'image. Les bords représentent les transitions de couleurs ou d'intensités entre les différentes zones de l'image.

### 2.6.2. Après l'application du filtre Sobel

Après l'application du filtre Sobel, l'image subit un processus de détection des bords. Ce filtre calcule les gradients d'intensité dans les directions X (horizontale) et Y (verticale) à l'aide de deux matrices de convolution. Le résultat de ce traitement est une image où les bords, c'est-à-dire les transitions de couleurs ou d'intensités, sont mis en évidence. Ainsi, les zones de l'image avec un changement brutal dans l'intensité des pixels sont accentuées, tandis que les zones avec des transitions douces ou homogènes restent pratiquement inchangées.

### 2.6.3. Ce que l'on doit attendre comme résultat

- **Image avec les bords mis en évidence :** Le filtre Sobel génère une représentation où les bords des objets présents dans l'image deviennent plus visibles. Les zones où il y a une transition marquée entre les couleurs ou les intensités (comme le bord entre deux objets) seront accentuées, tandis que les zones homogènes ou sans changement de couleur resteront presque inchangées.
- **Quatre sections de l'image traitées indépendamment :** L'image est divisée en quatre parties égales, et chaque section est traitée par l'un des quatre MicroBlaze. Chaque MicroBlaze applique le filtre Sobel sur sa propre section de l'image. Par conséquent, le résultat pour chaque section traitée doit être une représentation de cette partie de l'image avec les bords mis en évidence. Les sections traitées sont ensuite recomposées pour former l'image finale.

- **Le résultat final :** Une fois que les quatre sections traitées sont combinées, l'image complète résultante montre les bords de tous les objets présents dans la scène. L'image traitée aura une représentation claire des contours des objets, et ces contours seront bien définis grâce au filtre Sobel. Les zones de transition entre les objets seront plus visibles, tandis que les zones homogènes ou sans changements significatifs de couleur apparaîtront plus douces.



FIGURE 2.5 : Avant et après le traitement de l'image

#### 2.6.4. Temps de traitement

Le temps nécessaire au système pour traiter les images par les deux Microblazes, après que chaque quart d'entre elles soit placé dans la mémoire partagée, est de 27 ms.

```

Problems Tasks Console Properties SDK Terminal
TCF Debug Virtual Terminal - ARM Cortex-A9 MPCore #0
Data reception completed
Division completed.
Sobel kernels initialized.
Processing Time: 27 ms
Processing complete. Results:
Start sending processed data...
Processed data reconstructed and ready to send.
All processed data has been sent successfully.

```

FIGURE 2.6 : Temps de traitement des images.

#### 2.7. Système multi-horloge

Dans ce système, une configuration d'horloges multiples a été implémentée pour gérer les différentes parties de la conception (Figures 2.7 et 2.8). Cela garantit que des modules spécifiques fonctionnent de manière optimale sous différentes fréquences, obtenant ainsi un équilibre entre performances et fonctionnalités du système.

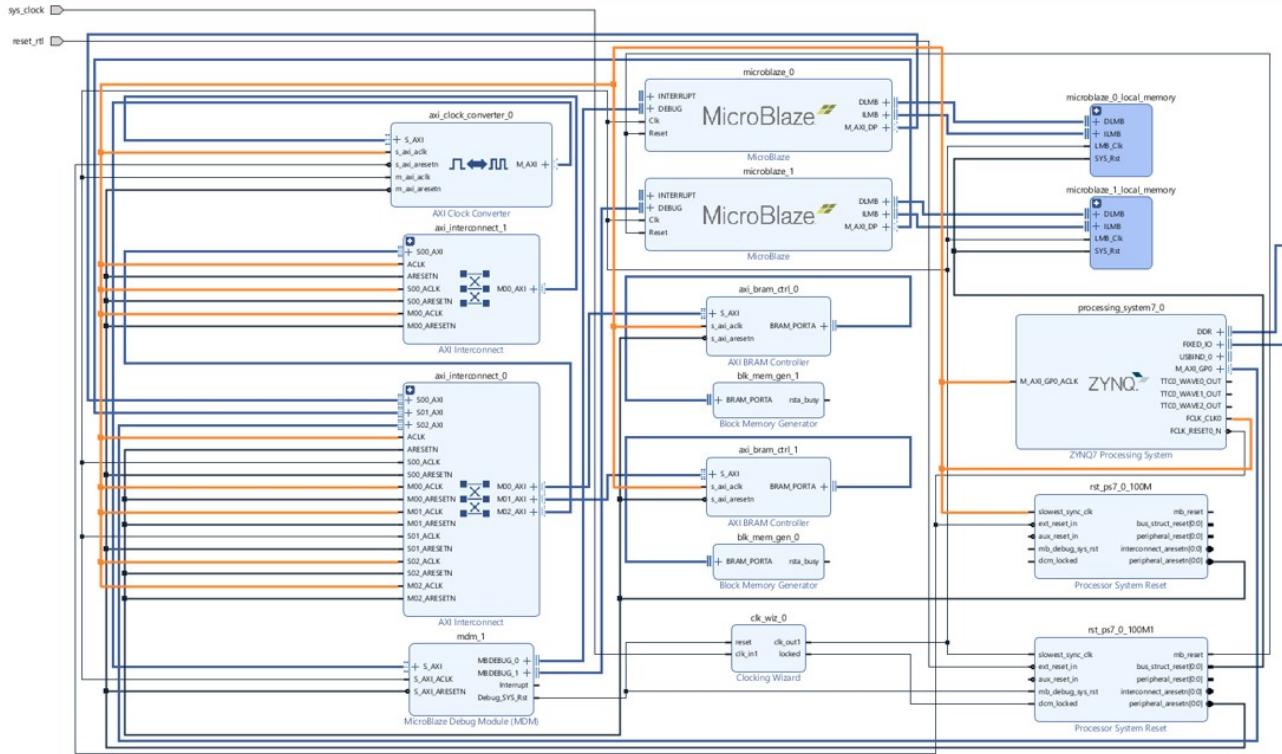


FIGURE 2.7 : Diagramme de blocs : Horloge # 1

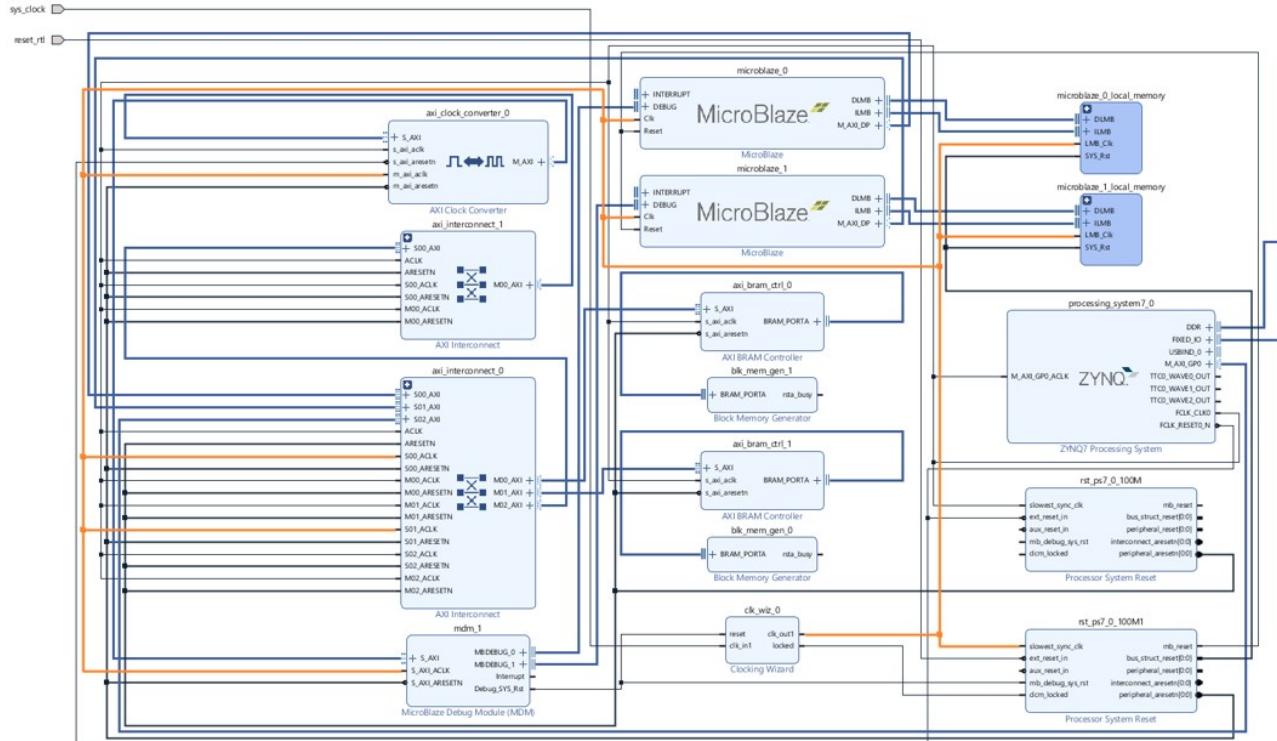


FIGURE 2.8 : Diagramme de blocs : Horloge # 2

Un **Clocking Wizard** a été utilisé pour générer et gérer deux fréquences d'horloge différentes dans le système. Ce bloc permet de dériver plusieurs signaux d'horloge à partir d'une source principale, en réglant chacun sur les fréquences souhaitées. Dans ce cas, l'assistant de pointage a fourni :

- Une horloge haute fréquence utilisée pour MicroBlaze, ses mémoires (ILMB, DLMB) et le AXI Memory Mapped Bus (MBA).
- Une horloge basse fréquence pour le reste des modules, comme l'ARM 9, la mémoire DDR et les interconnexions AXI.

Pour garantir une synchronisation appropriée entre les modules fonctionnant sous différents domaines d'horloge, un **AXI Clock Converter** a été inclus. Ce bloc facilite la communication entre des périphériques ou des blocs qui fonctionnent avec des fréquences d'horloge différentes au sein d'une interface AXI. Dans cette conception, l'AXI Clock Converter garantissait un transfert de données correct entre le MicroBlaze (avec son domaine d'horloge spécifique) et les modules contrôlés par le ZYNQ.

Les avantages obtenus avec ce type de configurations sont que chaque module fonctionne à la fréquence qui correspond le mieux à ses besoins fonctionnels, maximisant ainsi l'efficacité sans compromettre la capacité de traitement. De plus, une conception modulaire et efficace est autorisée, adaptée aux besoins spécifiques de chaque composant.

### 3. Q3 : Multicoeur 5 coeurs : 1 ARM et 4 MicroBlaze

#### 3.1. Flux de travail

Comme précédemment, le flux de traitement du système est décrit (Figure 3.1), cependant dans ce scénario, un système à 5 coeurs est utilisé avec un processeur ARM 9 et 4 processeurs Microblaze.

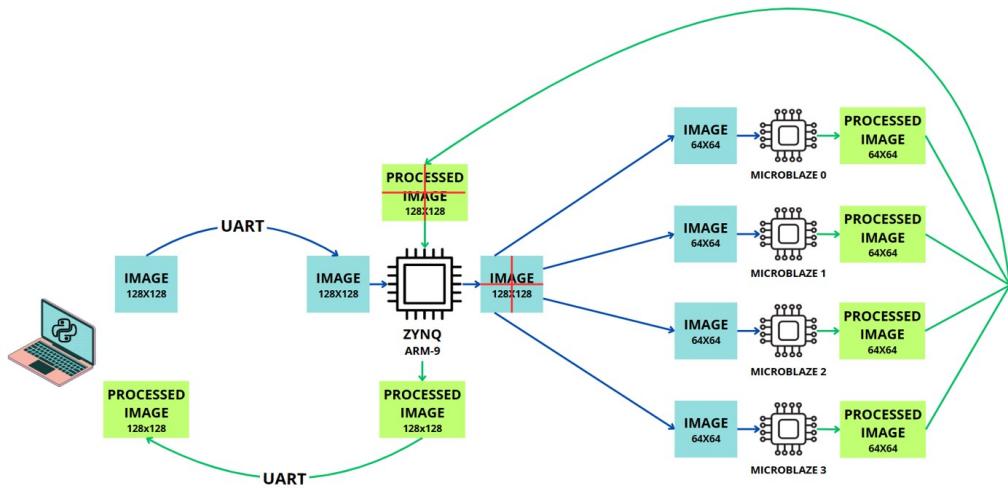


FIGURE 3.1 : Schéma explicatif du flux de travail du système.

L'envoi de l'image entre l'ordinateur et l'ARM est le même que précédemment, en utilisant l'interface UART.

De la même manière, lors de la réception de l'image complète, l'ARM 9 la divise en quatre sous-images de dimensions égales. Cependant, cette fois la répartition des sous-images se fait en tenant compte des deux nouveaux processeurs.

- MicroBlaze 0 : Traite la sous-image correspondant au premier quart de l'image, située dans le coin supérieur gauche.
- MicroBlaze 1 : Responsable de la sous-image correspondant au deuxième quart de l'image, située dans le coin supérieur droit.
- MicroBlaze 2 : Traite la sous-image correspondant au troisième quart de l'image, situé dans le coin inférieur gauche.
- MicroBlaze 3 : Il est responsable de la sous-image correspondant au quatrième et dernier quart de l'image, située dans le coin inférieur droit.

Les étapes restantes du processus sont effectuées de manière identique à ce qui a été décrit pour le système à 3 processeurs décrit ci-dessus.

### 3.2. Diagramme de blocs

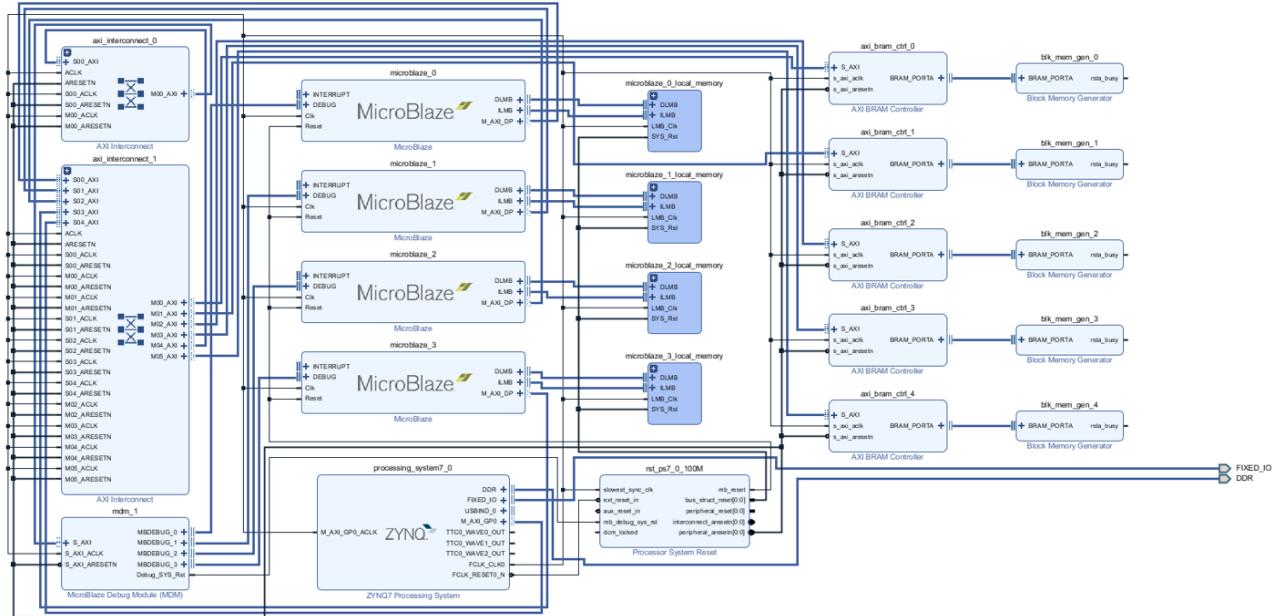


FIGURE 3.2 : Répartition de la mémoire.

Le système est conçu avec un **ZYNQ7 Processing System** et **quatre processeurs MicroBlaze** pour les tâches de traitement d'images. Le ZYNQ7, avec ses deux coeurs ARM Cortex-A9, agit comme l'unité centrale, gérant les tâches générales, la communication avec les périphériques, et le transfert de données vers la mémoire DDR et les blocs logiques programmables.

#### 3.2.1. Processeurs MicroBlaze

Le système inclut désormais **quatre processeurs MicroBlaze**, chacun étant affecté à une partie spécifique du traitement d'image. Ces processeurs fonctionnent en parallèle, ce qui améliore considérablement les performances du système. Chaque MicroBlaze est responsable du traitement d'une section distincte de l'image, en appliquant le filtre Sobel pour détecter les contours. Ce traitement parallèle permet au système de gérer des volumes de données plus importants et d'accélérer le traitement des images.

#### 3.2.2. Mémoire Locale pour les Processeurs MicroBlaze

Pour assurer un traitement rapide et efficace, chaque processeur MicroBlaze dispose de sa propre **mémoire locale**, spécifiquement le **DLMB** (Data Local Memory) et le **ILMB** (Instruction Local Memory). Cela permet à chaque MicroBlaze de fonctionner de manière autonome, avec un accès rapide aux données et instructions nécessaires pour le traitement en temps réel. L'ajout de deux mémoires locales supplémentaires fournit davantage de ressources dédiées pour les nouveaux processeurs MicroBlaze, permettant ainsi une optimisation et une parallélisation plus poussées de la charge de travail.

#### 3.2.3. BRAM et Contrôleurs

Le système utilise des **Block RAMs (BRAMs)**, gérées par les **AXI BRAM Controllers**, pour stocker les fragments d'image et les résultats intermédiaires pendant le processus de détection des bords. Avec l'ajout de **trois contrôleurs BRAM supplémentaires** et **blocs de mémoire BRAM**, le système a augmenté sa capacité de stockage temporaire. Ces nouvelles BRAMs agissent comme des tampons pour chaque processeur MicroBlaze, réduisant la latence et garantissant un débit élevé pendant le traitement de grandes quantités de données d'image. La structure BRAM élargie permet aux processeurs de stocker davantage de données intermédiaires localement, accélérant ainsi le processus global.

### 3.2.4. AXI Interconnects

Les **AXI Interconnects** servent de colonne vertébrale pour la communication, facilitant les transferts de données entre le ZYNQ7, les processeurs MicroBlaze, les mémoires locales, les contrôleurs BRAM et les autres périphériques. Avec l'ajout de plus de BRAM et de processeurs MicroBlaze, les interconnexions sont cruciales pour gérer l'augmentation du flux de données. Elles garantissent que chaque composant reçoive rapidement et efficacement les données dont il a besoin, soutenant le débit élevé nécessaire pour le traitement d'images en temps réel.

### 3.2.5. Débogage des MicroBlaze et Réinitialisation du Système

Le **MicroBlaze Debug Module** permet la surveillance et le débogage en temps réel des processeurs MicroBlaze, garantissant que toutes les tâches de traitement sont effectuées correctement. La **réinitialisation du système processeur** assure que le système démarre de manière synchronisée, évitant tout problème d'initialisation et maintenant la stabilité du design.

### 3.2.6. Vue d'Ensemble du Système

En résumé, le design exploite la puissance du ZYNQ7 pour le traitement général, la flexibilité des quatre processeurs MicroBlaze pour les tâches spécifiques de traitement d'images, la capacité améliorée des mémoires locales et des BRAM pour un traitement efficace des données, et les interconnexions AXI pour un flux de données fluide entre tous les composants. Ce système étendu est capable de gérer des tâches de traitement d'images plus complexes, avec une parallélisation et une efficacité accrues, ce qui en fait une solution idéale pour des applications en temps réel où la rapidité et la capacité à traiter de grands volumes de données sont cruciales.

## 3.3. Mémoires

De la même manière que précédemment, le système implémente une structure de mémoire hiérarchique conçue pour optimiser l'interaction entre quatre processeurs MicroBlaze et un système Zynq basé sur un ARM Cortex-A9. Ce schéma organise et segmente la mémoire pour diviser efficacement la charge de travail entre différents éléments, facilitant ainsi le traitement parallèle et une gestion efficace des ressources.

Par rapport au cas précédent, deux mémoires locales ont été ajoutées pour chaque microblaze et deux mémoires partagées supplémentaires, qui permettent la coordination entre les MicroBlaze et facilitent le transfert de données entre les différents processeurs.

La taille de la mémoire est restée à 32 Ko et deux nouveaux espaces mémoire ont été ajoutés (0x44000000 - 0x44007fff et 0x46000000 - 0x46007fff). La nouvelle distribution peut être revue dans la figure ??.

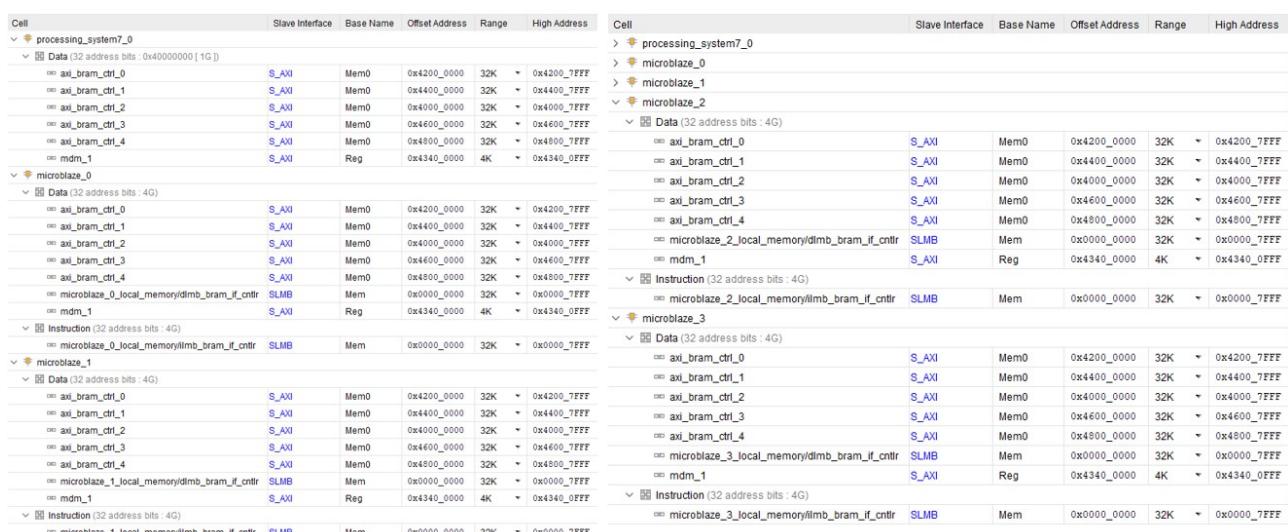


FIGURE 3.3 : Répartition de la mémoire.

De plus, le système de paralysie proposé précédemment a été maintenu, avec écrasement des masques ( $G_x$  et  $G_y$ ).

## 3.4. Résultats

### 3.4.1. Temps de traitement

Le temps nécessaire au système pour traiter les images par les quatre Microblazes, après que chaque quart d'entre elles soit placé dans la mémoire partagée, est de 13 ms.

Si l'on compare cela avec le résultat obtenu précédemment (27 ms), on constate que c'est la moitié du temps précédent. Ce résultat est conforme à ce que l'on attend de ce système, puisqu'il dispose de deux fois plus de processeurs en charge du traitement de l'image, ce qui parallélise complètement la tâche et réduit donc cette valeur.

```
Problems Tasks Console ✘ Properties SDK Terminal
TCF Debug Virtual Terminal - ARM Cortex-A9 MPCore #0
Data reception completed.
Sobel kernels initialized.
Processing Time: 13 ms
Processing complete. Results:
Start sending processed data...
Processed data reconstructed and ready to send.
All processed data has been sent successfully.
```

FIGURE 3.4 : Temps de traitement des images.

## 4. Q4 : Exploration Automatisée

Vivado est un outil complexe et complet, riche en méthodes d'optimisation qui peuvent être utilisées. Cependant, ces méthodes doivent être explorées, car leurs différences peuvent être significatives. Pour cela, il est nécessaire de réaliser une exploration automatique de différentes configurations d'optimisation afin de déterminer la meilleure configuration pour le projet.

### 4.1. Data Generation

On a commencé par la génération de données à l'aide d'un algorithme écrit en TCL pour automatiser la manipulation de Vivado. Cet algorithme a été utilisé pour générer des données issues de différentes configurations d'optimisation afin de comparer les résultats obtenus.

Cet algorithme était ajouté au dossier du projet à optimiser et exécuté via le Vivado TCL Shell. Il ouvrait le projet, effectuait les modifications nécessaires des configurations, puis relançait les différentes étapes du processus de simulation.

**Remarque.** On a opté pour l'utilisation de l'interface en ligne de commande, car elle s'avère plus efficace pour l'exécution de multiples commandes, étant donné qu'il n'est pas nécessaire d'effectuer des modifications ou des mises à jour de l'interface graphique.

De cette manière, les fichiers générés par Vivado étaient constamment réécrits, ce qui, d'une part, permettait de réduire la quantité finale de données à stocker, mais impliquait également qu'il était nécessaire de sauvegarder les fichiers requis pour une analyse ultérieure ainsi que pour permettre l'exécution sur carte.

Avant qu'une nouvelle simulation ne soit réalisée, les fichiers suivants étaient copiés vers un dossier externe contenant les caractéristiques de la simulation :

1. utilization placed
2. timing summary routed
3. power rated
4. bitstream
5. exported hardware

#### 4.1.1. Software Optimizations

Les optimisations logicielles prenaient en compte les configurations de Vivado indépendantes du projet en lui-même et concernaient exclusivement les méthodes de synthèse et d'implémentation utilisées. Voici ci-dessous quelques-unes des méthodes de synthèse et d'implémentation disponibles dans Vivado 2019.1 :

1. **Synthesis**
  - (a) Flow\_AreaOptimized\_high
  - (b) Flow\_PerfOptimized\_high
  - (c) Flow\_PerfThresholdCarry
  - (d) Flow\_RuntimeOptimized
  - (e) Vivado Synthesis Defaults
2. **Implementation**
  - (a) Area\_Explore
  - (b) Congestion\_SSI\_SpreadLogic\_high
  - (c) Flow\_Quick
  - (d) Performance\_Explore
  - (e) Power\_ExploreArea

#### (f) Vivado Implementation Defaults

Tous les méthodes ne sont pas listées ci-dessus car, après des tentatives initiales, il a été constaté qu'il n'y avait pas de grandes variations entre certaines méthodes appartenant à la même catégorie d'optimisations. Ainsi, il a été décidé de réduire l'espace de recherche aux méthodes présentant les variations les plus significatives, afin de concentrer les efforts computationnels sur celles-ci.

## 4.2. Data Analysis

Après l'exécution de l'algorithme de simulation, un script en Python a été développé pour extraire les données des fichiers générés par Vivado et les stocker dans une base de données sous forme de fichiers CSV. Cette base de données a été utilisée pour générer des graphiques permettant l'analyse des données obtenues.

### 4.2.1. Database Creation

Les fichiers CSV suivants ont été générés avec les caractéristiques ci-dessous :

1. **logs.csv** :

- (a) strategy\_synthesis
- (b) strategy\_implementation
- (c) datetime\_execution
- (d) duration\_execution
- (e) duration\_synthesis
- (f) duration\_implementation

2. **powers.csv** :

- (a) strategy\_synthesis
- (b) strategy\_implementation
- (c) power\_total
- (d) power\_dynamic
- (e) power\_static
- (f) power\_clocks
- (g) power\_logic
- (h) power\_signals
- (i) power\_block\_RAM
- (j) power\_IO
- (k) power\_PS7
- (l) temperature\_ambient\_max
- (m) temperature\_junction
- (n) confidence

3. **timings.csv** :

- (a) strategy\_synthesis
- (b) strategy\_implementation
- (c) WNS
- (d) TNS
- (e) TNS\_endpoints\_failing
- (f) TNS\_endpoints\_total
- (g) WPWS
- (h) TPWS
- (i) TPWS\_endpoints\_failing
- (j) TPWS\_endpoints\_total

4. **utilizations.csv** :

- (a) strategy\_synthesis
- (b) strategy\_implementation
- (c) LUT\_logic

- (d) LUT\_memory
- (e) slice\_registers\_FF
- (f) block\_RAM
- (g) IOB
- (h) BUFGCTRL

**Remarque.** Tous les résultats extraits des fichiers Vivado ne sont pas présentés dans ce rapport. Seuls ceux jugés les plus importants pour l'analyse ont été mis en avant.

Dans ce cas, une combinaison des stratégies de synthèse et d'implémentation a été utilisée comme clés d'identification pour les différents fichiers.

## 4.3. Résultats Q2

### 4.3.1. Duration

L'analyse commence par les temps d'exécution pour chaque combinaison de synthèse et d'implémentation, comme illustré ci-dessous :

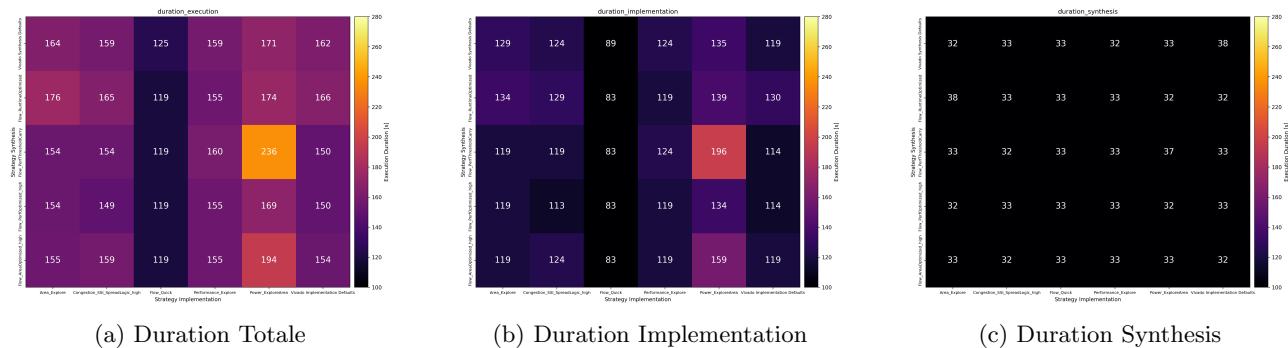


FIGURE 4.1 : Temps d'Exécution de Simulation pour Synthèse et Implémentation

**Remarque.** Il faut remarquer que le plus petit le valeur, le mieux.

**Remarque.** Il faut remarquer que la Figure 4.1a est égalé à la somme des Figures 4.1b et 4.1c.

Il est constaté qu'une stratégie d'implémentation présente systématiquement les temps d'exécution les plus courts, la "Flow\_Quick", qui, quel que soit le méthode de synthèse utilisée, affiche les temps les plus bas. Il est également à noter que la stratégie d'implémentation avec les temps d'exécution les plus élevés est la "Power\_ExploreArea", avec la méthode "Flow\_PathThresholdCarry" présentant une durée nettement plus longue que toutes les autres. Les autres combinaisons de méthodes ne montrent pas de résultats remarquables.

En analysant les graphiques indépendants des méthodes d'implémentation et de synthèse, on remarque que le principal facteur influençant le temps d'exécution est le temps d'implémentation, car les temps de synthèse sont essentiellement identiques pour toutes les combinaisons.

### 4.3.2. Power

Dans la séquence suivante, l'analyse porte sur les pertes de puissance simulées pour les circuits pour chaque combinaison de synthèse et d'implémentation, comme montré ci-dessous :

**Remarque.** Il faut remarquer que le plus petit le valeur, le mieux.

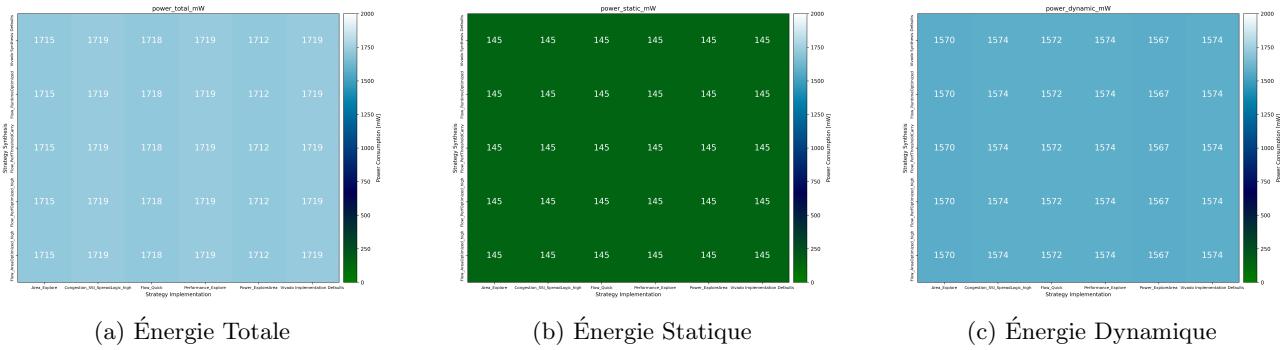


FIGURE 4.2 : Dissipacition d'Énergie de Simulation pour Synthèse et Implémentation

**Remarque.** Il faut remarquer que la Figure 4.2a est égalé à la somme des Figures 4.2b et 4.2c.

Il est à noter que, globalement, les valeurs sont pratiquement identiques et qu'il n'y a pas de grande variabilité entre les différentes valeurs.

Ainsi, une analyse plus détaillée a été effectuée sur les configurations qui ont donné les meilleures et les pires résultats en termes de temps d'exécution :

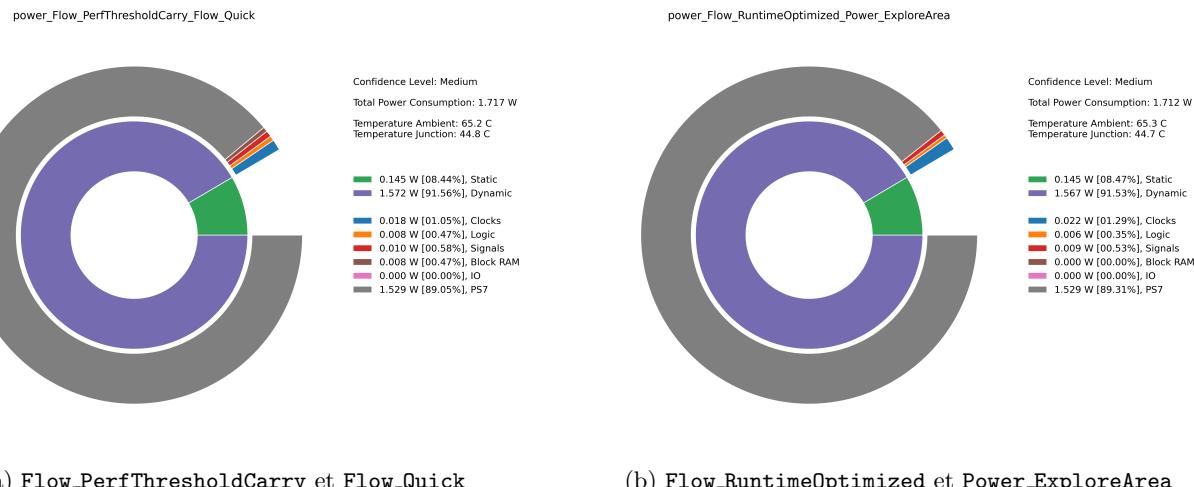


FIGURE 4.3 : Détail de Dissipation d'Énergie

Il n'y a pas de grande variabilité entre les différentes combinaisons et, de ce fait, on conclut que la puissance dissipée est peu influencée par les méthodes d'implémentation ou de synthèse utilisées.

### 4.3.3. WNS

On passe maintenant à l'analyse du "Worst Negative Slack (WNS)" pour les différentes combinaisons de synthèse et d'implémentation, comme démontré ci-dessous :

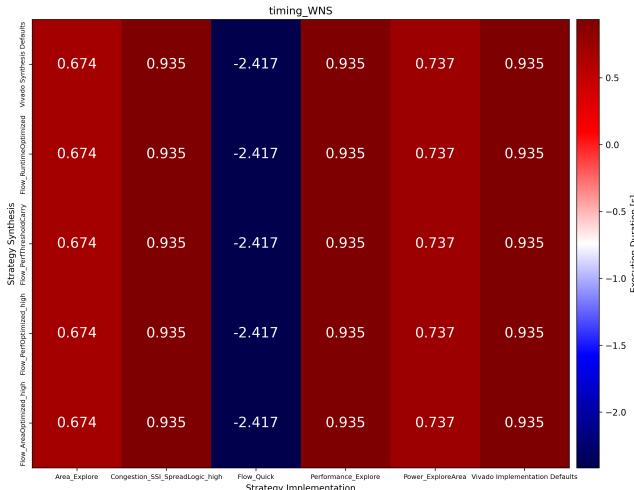


FIGURE 4.4 : WNS Timing pour Synthèse et Implémentation

**Remarque.** Il faut remarquer que le plus proche de zéro, le mieux. En cas négatif, sous-optimale.

Ici, il devient évident que le compromis est fait entre le temps d'exécution et le WNS. La combinaison de "Flow\_PerfThresholdCarry" et "Power\_ExploreArea" présente la deuxième meilleure performance de WNS, tandis que la combinaison de "Flow\_RuntimeOptimized" et "Flow\_Quick" présente la pire performance de WNS, ce qui démontre que l'exécution a été plus courte car une solution n'a pas été trouvée.

Il est important de souligner que le choix de la stratégie de synthèse n'a pas influencé le résultat du WNS, étant donné que la stratégie d'implémentation est le principal facteur déterminant du résultat.

#### 4.3.4. Ressources

Enfin, l'analyse de l'utilisation des ressources pour les différentes combinaisons de synthèse et d'implémentation est effectuée comme démontré ci-dessous :

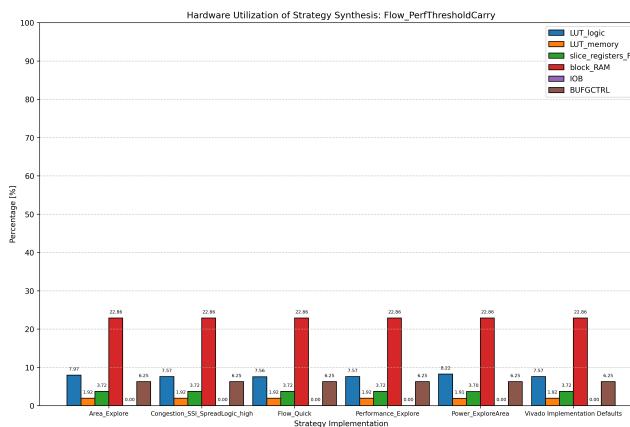


FIGURE 4.5 : Utilisation de Ressources pour Flow\_PerfThresholdCarry

**Remarque.** Il faut remarquer que le plus petit, le mieux.

Il n'y a pas de variations significatives dans l'utilisation des ressources matérielles entre les différentes configurations d'implémentation.

## 4.4. Résultats Q3

### 4.4.1. Duration

L'analyse commence par les temps d'exécution pour chaque combinaison de synthèse et d'implémentation, comme indiqué ci-dessous :

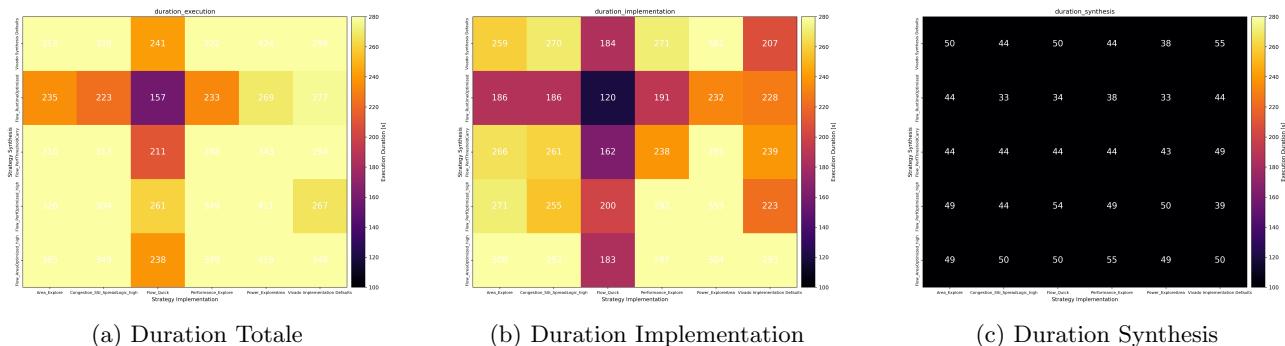


FIGURE 4.6 : Temps d'Exécution de Simulation pour Synthèse et Implémentation

**Remarque.** Il faut remarquer que le plus petit le valeur, le mieux.

**Remarque.** Il faut remarquer que la Figure 4.6a est égalé à la somme des Figures 4.6b et 4.6c.

Il est à noter que, comme expliqué pour la question Q2, la stratégie d'implémentation "Flow\_Quick" présente les temps d'exécution les plus bas, tandis que la stratégie de synthèse "Flow\_RuntimeOptimized" donne des résultats légèrement supérieurs.

On observe également que la stratégie d'implémentation avec les temps d'exécution les plus élevés est "Power\_ExploreArea", avec la méthode "Flow\_RuntimeOptimized" affichant une durée significativement plus longue que toutes les autres. Les autres combinaisons de méthodes ne montrent pas de résultats remarquables.

Comme la même échelle a été utilisée pour les deux cas, il est évident que les temps d'exécution sont généralement plus élevés dans ce cas-ci, ce qui est attendu, étant donné que ce circuit présente une architecture nettement plus sophistiquée.

### 4.4.2. Power

Ensuite, l'analyse porte sur la puissance dissipée simulée pour les circuits, en fonction de chaque combinaison de synthèse et d'implémentation, comme indiqué ci-dessous :

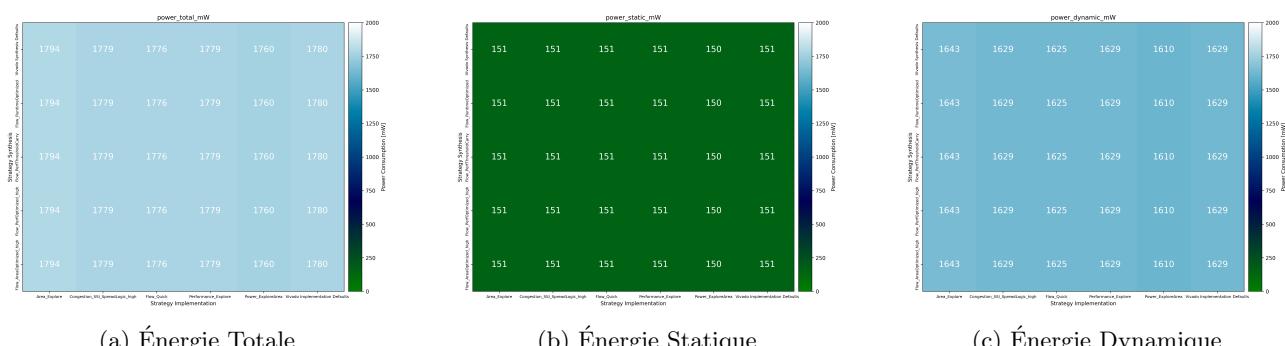


FIGURE 4.7 : Dissipacition d'Énergie de Simulation pour Synthèse et Implémentation

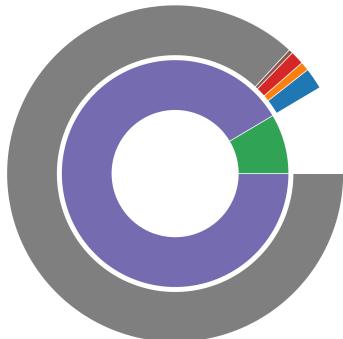
**Remarque.** Il faut remarquer que le plus petit la valeur, le mieux.

**Remarque.** Il faut remarquer que la Figure 4.7a est égale à la somme des Figures 4.7b et 4.7c.

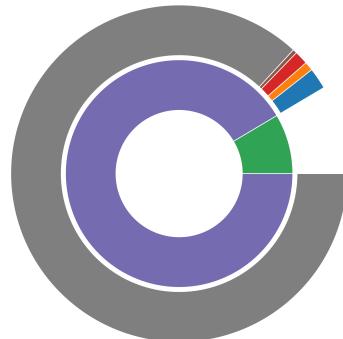
Il est à noter que, comme pour la question Q2, les valeurs sont pratiquement identiques et qu'il n'y a pas de grande variabilité entre les différentes valeurs.

Ainsi, une analyse plus détaillée a été effectuée sur les configurations qui ont donné les meilleures et les pires résultats en termes de temps d'exécution :

power\_Flow\_PerfThresholdCarry\_Power\_ExploreArea



power\_Flow\_RuntimeOptimized\_Power\_ExploreArea



(a) Flow\_PerfThresholdCarry et Power\_ExploreArea

(b) Flow\_RuntimeOptimized et Power\_ExploreArea

FIGURE 4.8 : Détail de Dissipation d'Énergie

Il n'y a pas de grande variabilité entre les différentes combinaisons, et ainsi, il est conclu que la puissance dissipée est peu influencée par les méthodes d'implémentation ou de synthèse utilisées.

#### 4.4.3. WNS

Nous passons maintenant à l'analyse du Worst Negative Slack (WNS) pour les différentes combinaisons de synthèse et d'implémentation, comme démontré ci-dessous :

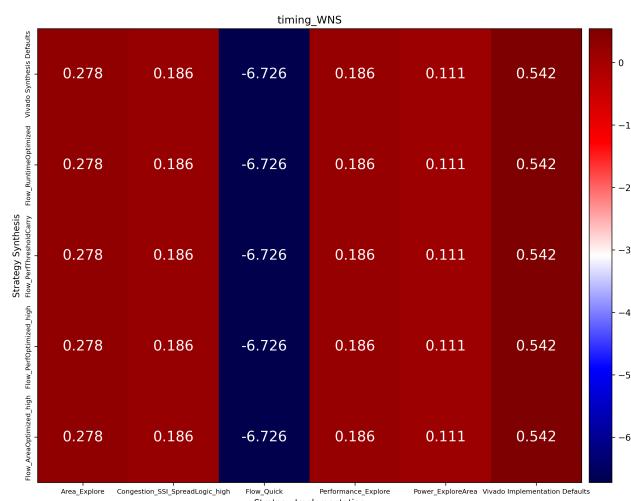


FIGURE 4.9 : WNS Timing pour Synthèse et Implémentation

**Remarque.** Il faut remarquer que le plus proche de zéro, le mieux. En cas négatif, sous-optimale.

Comme observé pour Q2, le pire résultat est obtenu avec la stratégie d'implémentation "Flow\_Quick" et le meilleur avec "PowerExploreArea". La différence entre les résultats est plus significative dans ce cas que dans le précédent, ce qui indique que la complexité de l'architecture implémentée a un impact important sur les gains, ou non, des méthodes d'optimisation.

#### 4.4.4. Ressources

Enfin, l'analyse de l'utilisation des ressources pour les différentes combinaisons de synthèse et d'implémentation est présentée ci-dessous :

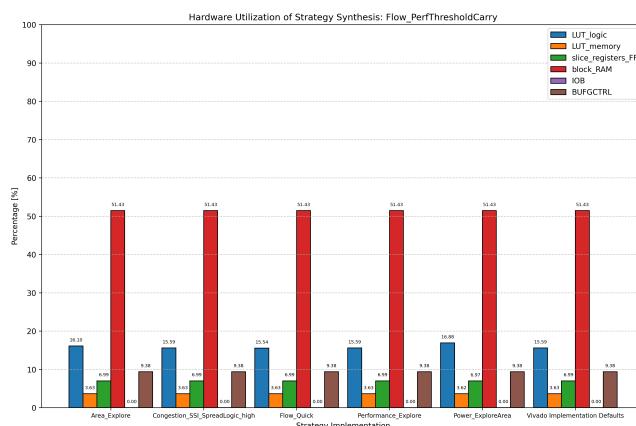


FIGURE 4.10 : Utilisation de Ressources pour Flow\_PerfThresholdCarry

**Remarque.** Il faut remarquer que le plus petit, le mieux.

Encore une fois, on remarque qu'il n'y a pas de grande variabilité entre les différentes options d'implémentation. Cependant, toutes les parties du matériel sont plus sollicitées dans ce cas que dans le cas précédent.

### 4.5. Résultats Généraux

En résumé, on peut observer que les résultats obtenus expérimentalement subissent de petites modifications lorsque l'on manipule des stratégies et des algorithmes d'optimisation. Bien que de petits gains puissent être obtenus en utilisant les outils disponibles dans Vivado, la capacité à concevoir une architecture efficace et optimisée influence beaucoup plus fortement le résultat final.

Ainsi, il est important de souligner l'importance de travailler avec soin lors de la conception des différentes parties du matériel et du logiciel afin d'obtenir les meilleurs résultats possibles avec le matériel disponible, car l'optimisation du logiciel et du matériel ne peut compenser une architecture mal conçue.