Partie

1 Contexte de l'étude

Cet enseignement de dernière année a pour objectif de vous familiariser avec les méthodologies de conception employées lors de la création de systèmes numériques « complexes ». Les systèmes complexes, nommés SoC et/ou SoPC intègrent actuellement des cœurs de processeurs ainsi que des accélérateurs matériels. Afin de mettre au point et d'exploiter efficacement de tels systèmes, il est indispensable de bénéficier de méthodologies et d'outils de haut-niveau d'abstraction pour les concevoir, les étudier et les implanter.

Cette séquence pédagogique sera illustrée à l'aide d'une application de compression d'image (JPEG). Malgré son âge avancé, ce type d'application de compression d'images est encore couramment utilisée dans bon nombre de systèmes embarqués actuels : appareils photos, téléphone, etc. De plus sa forme modulaire rend son étude accessible.

De nombreuses ressources traitant de la compression d'images existent sur Internet. En conséquence, je vous demande de jeter un coup d'œil et/ou lire une partie des liens suivants :

Image Compression - How Math Led to the JPEG2000 Standard
Image Compression: Seeing What's Not There
https://fr.wikipedia.org/wiki/JPEG
https://en.wikipedia.org/wiki/JPEG - encoding

La séquence pédagogique qui vous est proposée se limite à l'étude de quelques blocs fonctionnels. Cependant, pour des raisons de simplicité, dans les parties suivantes, nous considérerons que vous maitrisez les concepts de base de la compression JPEG. En cas de besoin n'héistez pas à vous vers les documentations proposées ci-dessus.

Partie

2 Etude de la conversion colorimétrique

1. INTRODUCTION

Avant de vous lancer dans la modélisation, le raffinement et l'implémentation du système de compression complet, vous allez apprendre à dompter le langage SystemC sur un sous-bloc de la chaîne JPEG. La première partie de votre travail consiste à modéliser et à étudier le premier bloc de la chaîne de compression. Ce bloc de « faible complexité calculatoire » est en charge du changement d'espace colorimétrique.

Le changement d'espace colorimétrique, de l'espace de couleur RGB (rouge, vert, bleu) vers l'espace YCbCr (luminance, chrominances) a pour objectif d'améliorer l'efficacité du processus de compression JPEG. En effet l'œil humain est peu sensible à l'altération de la chrominance.

La conversion entre les 2 espaces colorimétriques est réalisée à l'aide des formules mathématiques fournies ci-dessous :

$$Y = 0.299 \times R + 0.587 \times G + 0.114 \times B$$
$$C_b = -0.1687 \times R - 0.3313 \times G + 0.5 \times B + 128$$
$$C_r = 0.5 \times R - 0.4187 \times G - 0.0813 \times B + 128$$

L'opération inverse qui convertit les informations de l'espace YCbCr vers l'espace RGB est réalisée à partir des équations fournies ci-dessous :

$$R = Y + 1.402 \times (C_r - 128)$$
$$G = Y - 0.34414 \times (C_b - 128) - 0.71414 \times (C_r - 128)$$
$$B = Y + 1.772 \times (C_b - 128)$$

Pour des raisons partiques, les valeurs des composantes {R, G, B} et {Y, Cb, Cr} sont représentées à l'aide de données entières et appartiennent à l'intervalle [0, 255].

Dans un premier temps vous allez implanter un modèle système réalisant juste ces conversions colorimétriques afin de vous familiariser avec les concepts de SystemC.

2. MODELE FONCTIONNEL SANS SYSTEMC

Vous allez commencer par écrire un programme en C/C++. Ce programme en C/C++ aura pour vocation de vous permettre de tester les 2 fonctions que vous allez écrire.

Ces 2 fonctions nommées respectivement RGB_2_YCrCb et YCbCr_2_RGB réaliserons les transformations définies dans les équations fournies précédemment.

Afin d'uniformiser vos différents développements et aussi simplifier votre travail durant ce TP, vous utiliserez les prototypes suivants pour vos deux fonctions :

```
void RGB_2_YCrCb (int rvb[3], int ycbcr[3]);
void YCbCr_2_RGB (int ycbcr[3], int rvb[3]);
```

Remarque

> Par convention, les éléments notés rvb[0], rvb[1], rvb[2] désigneront respectivement les canaux rouge, vert et bleu. De manière analogue, les éléments notés ycbcr[0], ycbcr[1], ycbcr[2] désigneront respectivement la luminance Y et les chrominances C_b et C_r.

A partir de ces informations :

- Écrivez les deux fonctions et votre programme \$main\$ dans un
- Testez votre programme à l'aide des triplets de données fournis ci-dessous.

```
RGB = {  0,   0,   0}
RGB = {  0,   0, 128}
RGB = {  0, 128,   0}
RGB = {128,   0,   0}
```

Cette première description algorithmique des fonctions de conversion sera le point de départ du raffinement des fonctions mathématiques vers une implantation VHDL.

3. ENCAPSULATION DANS UN MODELE SYSTEMC

A partir de maintenant nous allons rentrer dans le vif du sujet. En effet, nous allons utiliser le langage SystemC afin de progressivement transformer vos codes source algorithmiques en codes VHDL synthétisables (ou tout au moins vers des versions s'en rapprochant).

Dans un premier temps, nous allons réécrire votre programme **main** sous la forme d'un modèle SystemC. Vos fonctions vont être incluses dans un modèle, tout comme la génération des données de test et l'analyse des résultats.

A vous de jouer :

- Récupérez les fichiers fournis par votre enseignant [URL].
- Décompressez les et modifiez les en ajoutant les fonctions C que vous avez précédemment écrites.
- Dessinez le modèle SystemC tel qu'il est décrit.
- Expliquez en quoi vous venez de réaliser une encapsulation de vos fonctions.
- Compilez votre programme à l'aide du makefile fourni puis lancez la simulation.

Vous venez de simuler votre premier système décrit en SystemC. Toutefois comme vous venez de vous en rendre compte le système que nous avons décrit n'est pas très pratique pour valider le fonctionnement de vos fonctions.

4. MODELISATION DU SYSTEME D'ACQUISITION DES IMAGES

Afin de se rapprocher de la modélisation d'un système réel, nous allons modifier notre description SystemC : nous allons maintenant travailler sur des images (flux de données) réelles. Pour faire évoluer votre modèle, il est seulement nécessaire de remplacer le générateur de données et l'analyseur présent en bout de chaine.

A vous de jouer :

- Copiez les fichiers cpp que vous avez modifiez à la question précédente dans le répertoire relatif à cette question.
- Ouvrez les modules en charge de la génération et de l'analyse des données.
- Dessinez le modèle SystemC tel qu'il est actuellement décrit.

- Compilez le nouveau modèle à l'aide du makefile fourni puis lancez la simulation.
- Vérifiez que l'image générée est conforme.
- Analysez les images présentant les différences entre les images et expliquez les.

Vous aurez remarqué à la fin de la simulation qu'un nombre important d'informations est affiché dans le terminal. Ces informations sont des métriques mathématiques permettant d'évaluer les dégradations induites par vos transformations sur l'image de sortie. Cela vous permettra dans les questions suivantes d'évaluer la pertinence de vos décisions (ou les bugs éventuels).

5. RAFFINEMENT DES INTERFACES DE COMMUNICATION (1)

Jusqu'à maintenant les communications au sein du système se sont fait en manipulant des données sur 32 bits (int). Toutefois ce format de données n'est pas celui utilisé dans le monde réel. En effet les informations de type RGB et YCbCr sont codées sur 8 bits.

Vous allez donc modifier les modèles SystemC de la question précédente afin refléter plus efficacement le comportement réel du système. Pour cela, vous pourrez utiliser à votre convenance soit le type **unsigned char** soit la classe SystemC prévue à cet effet **sc_uint<T>**.

A vous de jouer :

- Dupliquez le répertoire de lié à la question précédente et renommez le en Part1_Question_4. Modifiez les fichiers afin d'incorporer ce raffinement des interfaces de communication.
- Compilez le modèle modifié puis lancez la simulation.
- Vérifiez que l'image générée est conforme à l'original.

Normalement, vous avez du découvrir que la simulation ne fourni plus les résultats attendus. Pour ceux qui ne l'auraient pas vu, deux solutions, votre modèle est faux (vous n'avez pas bien appliqué les modifications demandées) soit vous n'avez pas validé correctement l'image de sortie.

A vous de rejouer :

- Corrigez vos algorithmes afin de résoudre le problème observé.
- Compilez le modèle corrigé puis lancez la simulation.
- Vérifiez que l'image générée est maintenant conforme.

Comme vous venez de le remarquer une modification insignifiante peut relever des problèmes réels. Imaginez la difficulté pour identifier et corriger ce genre de problème lorsque l'on se trouve en VHDL et que l'on manipule des vecteurs de bits.

6. CONCLUSION

Maintenant que vous avez un modèle fonctionnel. Vous allez remplacer les calculs flottants réalisés dans les modules de conversion par des calculs en virgule fixe. Cependant avant d'entreprendre cette tache au combien fastidieuse, vous allez faire un peu de VHDL de manière à vous convaincre de l'intérêt de cette transformation.

Chapitre 3

Opérateurs flottants sur cible FPGA

1. INTRODUCTION

Afin de vous convaincre des performances calamiteuses du format de codage flottant et par voie de conséquence du bien fondé du codage en virgule fixe, vous allez faire quelques synthèses logiques… Vous allez successivement observer l'évolution des ressources matérielles mise en œuvre pour réaliser différents types d'opérations arithmétiques manipulant des données entières et flottantes. L'évaluation que vous allez effectuer sera basée sur la technologie FPGA. La famille Virtex-7 (XXXX) de Xilinx sera considérée et l'outil de synthèse employé sera Xilinx Vivado.

2. ADDITION DE DEUX NOMBRES ENTIERS SIGNES

La première série d'expériences va cibler l'évaluation des caractéristiques des opérations d'addition entre des nombres entiers signés. Pour cela, vous allez utiliser un additionneur asynchrone (aucun registre ne sera ajouté en amont et en aval de l'opérateur).

Remarque

> Pour des raisons de simplicité, nous omettrons la propagation de la retenue. Ainsi l'opérateur d'addition que vous devez implanter consomme deux données sur N bits en entrée et produit une donnée sur N bits en sortie. Dans un processeur traditionnel (ARM, INTEL, etc.), ce ne serait pas le cas.

A partir de ces informations :

- Écrivez le module VHDL qui implante cet additionneur. Vous prendrez N=8 pour commencer.
- Réalisez la synthèse logique de votre module. Noter la complexité matérielle de l'opérateur (nombre de LUTs) ainsi que la durée du

chemin critique. Dans les options de synthèse décochez la case « Add IO buffers » afin de supprimer l'insertion des pattes d'E/S du calcul du chemin critique.

- Faire varier la largeur des mots d'entrée avec les valeurs suivantes : 16 bits, 24 bits, 32 bits et 64 bits.

Remarque

> Notez dans un tableau les résultats de synthèse obtenus pour chaque largeur de mots. Vous pouvez représenter les résultats sous forme de courbes.

3. MULTIPLICATION DE DEUX NOMBRES ENTIERS SIGNES

L'opérateur d'addition est l'opérateur le plus simple. Dans la majorité des applications, d'autres types d'opération sont nécessaires. Vous allez maintenant étudier le second opérateur le plus employé : la multiplication.

Comme cela a été réalisé dans la première partie, vous allez étudier un multiplieur asynchrone manipulant des nombres entiers signés.

Remarque

> Pour des raisons de simplicité, nous omettrons la propagation des bits de poids fort en sortie de la multiplication. En effet une multiplication de deux données codées sur N bits produit un résultat sur 2N bits. Ainsi l'opérateur de multiplication que vous devez implanter consomme deux données sur N bits en entrée et produit une donnée sur N bits en sortie. Dans un processeur traditionnel ce n'est toujours le cas.

A partir de ces informations :

- Écrivez le module VHDL qui implante cette multiplication. Vous prendrez N=8 pour commencer.
- Réalisez la synthèse logique de votre module. Noter la complexité matérielle de l'opérateur (nombre de LUTs) ainsi que la durée du chemin critique. Attention, vous prendrez soin dans un premier temps de désactiver l'utilisation des DSP Blocks dans les options de synthèse logique.
- Faire varier la largeur des mots d'entrée avec les valeurs suivantes : 16 bits, 24 bits, 32 bits et 64 bits.
- Réactivez l'utilisation des DSP Blocks dans les options de synthèse logique et recommencez la campagne de mesure.

- Interprétez et comparez les résultats que vous venez d'obtenir et comparez les avec ceux obtenus sans DSP Blocks. Pour cela, il vous est conseillé de jeter un coup d'œil aux documents suivants : Cours de FPGA 2A, Xilinx UG479.
- Comparez les résultats obtenus avec ceux des additionneurs synthétisés précédemment.

4. MULTIPLICATION D'UN NOMBRE PAR UNE CONSTANTE

Vous allez maintenant étudier un cas particulier. Vous allez implanter quelques multiplications par des valeurs constantes afin de voir comment l'outil de synthèse les gère.

- Ecrivez un premier multiplieur 32 bits qui multiplie une entrée codée sur N bits avec une constante codée sur 10 bits. La constante aura pour valeur 8. Désactivez l'utilisation des DSP Blocks.
- Réalisez la synthèse logique de votre module. Interprétez les résultats obtenus.
- Refaites les mêmes opérations lorsque la constante vaut 261.

5. DIVISION DE 2 NOMBRES ENTIERS

Vous allez terminer l'étude des opérateurs entiers avec l'étude de l'opérateur de division. Cette opération peut s'implanter sous la forme. Cette étude se limitera à l'opérateur de division 32 bits manipulant des données non signées.

- Dessinez la structure de deux étages « élémentaires » de l'opérateur de division.
- Récupérer le diviseur 32 bits que votre enseignant a mis à votre disposition afin de vous faire gagner beaucoup de temps.
- Réalisez la synthèse logique du module de division. Interprétez le résultat obtenu.

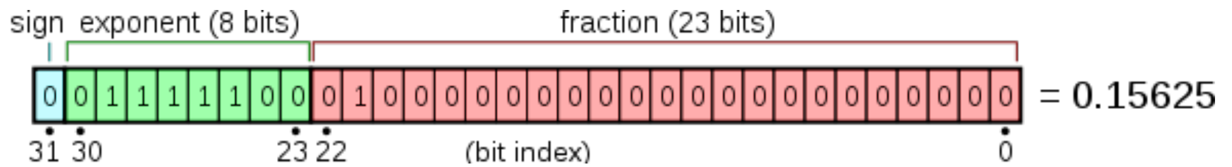
6. SYNTHESE PRELIMINAIRE DES DONNEES RECOLTEES

Vous venez de récolter un nombre important de mesures. Il est temps d'en faire une synthèse… Positionnez les informations de complexité, latence des différents opérateurs les uns par rapport aux autres. Essayer d'en tirer des enseignements.

7. MULTIPLICATION DE 2 NOMBRES FLOTTANTS (32 BITS)

Maintenant que vous maitrisez l'implantation des opérations entières, vous allez étudier les opérateurs flottants. Pour commencer, vous allez essayer de comprendre la manière dont est réalisé une multiplication

flottante entre deux nombres. Le format de codage des nombres flottants sur 32 bits est décrit dans la figure suivante :



Pour vous aider dans votre travail vous avez à votre disposition le code VHDL décrivant l'opération de multiplication disponible ici ainsi que divers documents disponibles sur internet.

- Analysez le code VHDL de l'opérateur de multiplication afin de retrouver la description algorithmique de l'opération.
- Dessinez l'architecture matérielle mise en œuvre dans le code VHDL.
- Faites une synthèse du multiplieur flottant afin d'estimer ses performances (slices, chemin critique).
- Comparez les performances relevées avec celles des multiplieurs entiers.

8. ADDITION DE DEUX NOMBRES FLOTTANTS

Maintenant que vous savez comment sont codés les nombres flottants et que vous maitrisez leur multiplication, nous allons maintenant étudiez le processus d'addition. Téléchargez le code VHDL décrivant l'opération d'addition disponible ici et retournez vous documenter sur le principe de base de l'adition flottante par exemple ici ou bien là (cours de guyot).

- Analysez le code VHDL de l'opérateur de multiplication afin de retrouver la description algorithmique de l'opération.
- Dessinez l'architecture matérielle mise en œuvre dans le code VHDL.
- Faites une synthèse de l'additionneur flottant d'extraire ses caractéristiques.
- Comparez les performances relevées avec celles obtenues précédemment.

9. AMELIORATION DES PERFORMANCES

Les performances de opérateurs flottants peuvent être améliorées en ajoutant par exemple des tranches de pipeline en interne (réduction des chemins critiques). Nous ne réaliserons pas cette étape fastidieuse manuellement dans le cadre de ce TP.

- Utilisez l'outil **Core Generator**, afin d'utiliser les composants matériels développés par les ingénieur de chez Xilinx. Développez

plusieurs solutions de multiplieurs et d'additionneurs afin de regarder les compromis surface & débits accessibles.

- Essayer de synthétiser au moins un opérateur afin de valider les estimations de performance fournies.
- L'utilisation de **Core Generator** est elle intéressantes vis à vis des opérations décrites manuellement ? Pour quelles caractéristiques ? Quel est le cout d'un tel outil ?

10. CONCLUSION ET SYNTHESE

Cette séquence pédagogique est maintenant terminée. Il ne vous reste plus qu'à comparer les différentes solutions que vous avez expérimentées dans le cadre de ce TP.

Chapitre 4 Raffinement de la conversion colorimétrique

1. INTRODUCTION

Maintenant que vous avez pleinement conscience du cout des opérateurs arithmétiques flottants et entiers sur cible FPGA, et indirectement sur cibl ASIC vous allez prendre « plaisir » à mettre en œuvre le codage en virgule fixe…

2. TRANSFORMATION EN VIRGULE FIXE DES CALCULS

Nous souhaitons concevoir une architecture matérielle permettant de réaliser la conversion RGB_2_YCrCb. Dans des systèmes ou les calculs flottants sont complexes à implanter et/ou onéreux. Pour cette raison, ces calculs manipulant des données flottantes doivent être transformés en calculs en virgule fixe. Vous allez donc procéder à cette transformation. On limitera cette étude à la fonction RGB_2_YCrCb.

Pour simplifier cette tache ardue, le langage SystemC propose un ensemble de classes dédiées à ce raffinement. Pour mener à bien votre tache, vous disposerez des classes suivantes :

```
sc_fixed <wl, iwl, q_mode, o_mode> var_signed;
sc_ufixed<wl, iwl, q_mode, o_mode> var_unsigned;

// EXAMPLE D'UTILISATION
float adder(float a, float b)
{
	sc_fixed<4,2,SC_RND,SC_WRAP> Inputa = a;
	sc_fixed<6,3,SC_RND,SC_WRAP> Inputb = b;
	sc_fixed<7,4,SC_RND,SC_WRAP> Output;
	Output = (Inputa + Inputb);
	return Output;
}
```

Les paramètres du template de classe ont les significations suivantes :

- **wl** : nombre total de bits dans le mot;
- **iwl** : nombre de bits alloué à la partie entière;
- **q_mode** : mode de quantification;
- **o_mode** : gestion de la saturation en cas de dépassement.

Afin de limiter la difficulté de cette tache que vous effectuez pour la première fois, nous considérons que tous les coefficients fractionnaires seront codés de la manière uniforme. Votre objectif est de trouver le bon format pour les coefficients et les opérations afin de limiter le bruit de calcul.

- Demandez des explications à votre enseignant si vous ne maîtriser pas les concepts de codage en virgule fixe.
- Dupliquez le répertoire de lié à la question précédente et renommez le en Part1_Question_5.
- Modifier la description de la fonction RGB_2_YCrCb pour la transformer en virgule fixe.
- Compilez le modèle puis lancez la simulation.
- Faites plusieurs tentatives afin de trouver un format permettant de limiter les pertes. Plus le format de codage est faible moins l'architecture matérielle sera onéreuse.

Vous avez du regarder les métriques mathématiques afin de vous guider lors de l'étape de raffinement. Toutefois, la qualité visuelle n'évolue pas de manière similaire à ces valeurs.

- Recommencer votre étude en considérant votre perception visuelle. Jusqu'à combien de bits pouvez-vous descendre avant de percevoir visuellement une dégradation ?

Vous venez de dimensionner la taille des données et des opérations contenues votre algorithme. Comme vous venez le constater un dimensionnement bien réaliser vous permet de gagner sur le cout silicium de votre architecture. Une fois de plus vous remarquerez qu'une telle étude est dure à réaliser en VHDL.

3. VIRGULE FIXE SANS UTILISER LES CLASSES SYSTEMC

Les classes **sc_fixed** et **sc_ufixed** sont très pratiques mais vous serez incapable d'écrire le VHDL équivalent. Afin de simplifier cette tache (l'écriture du VHDL) vous allez traduire votre codage en virgule fixe utilisant les classes **sc_fixed** par du code manipulant des **sc_int<T>** ou **sc_uint<T>**. Ces classes vous permettent d'avoir un niveau de description identique à celui que vous avez avec les type SIGNED et UNSIGNED en VHDL.

- Dupliquez le répertoire de lié à la question précédente et renommez le en Part1_Question_6.
- Modifiez la description de la fonction RGB_2_YCrCb pour ne plus utiliser que des type **sc_int<T>** et **sc_uint<T>**.
- Compilez le modèle puis lancez la simulation.
- Validez les performances de votre modèle par rapport à celui qui utilisait des **sc_fixed** et **sc_ufixed**.

Vous avez maintenant un code VHDL dont le niveau d'abstraction est identique a celui du VHDL. Vous remarquerez qu'une traduction de l'un à l'autre n'est pas très complexe.

4. RAFFINEMENT DES INTERFACES AU CYCLE-PRES

Maintenant que le comportement interne du composant a été étudié, nous allons raffiner les interfaces de communication du module.

Le système dans lequel votre composant sera intégré produit un triplet RGB tout les 30 ns. Ces données sont produites et transmises en même temps à votre composant. Ces données seront valides 10 ns et durant ce temps, un signal data_valid sera positionné à 1 (et 0 les 20 ns restantes).

Le comportement des sorties de votre composant devra être similaire: les sorties se feront sur 24 bits durant 10 ns. Un signal out_valid validera la présence de nouvelles sorties durant 10 ns.

Le temps de calcul entre la réception des données dans votre composant et la production des sorties sera fixé à 20 ns.

Afin de ne pas modifier le générateur de données ni le module de conversion YCrCb_2_RGB, la méthode la plus simple consiste à développer 2 wrappers permettant de transformer les liens actuels au protocole défini ci-dessus. De plus pour synchroniser l'ensemble des composants, une horloge de type sc_clock sera nécessaire.

- Dessinez le système que vous devez modéliser.
- Dupliquez le répertoire de lié à la question précédente et renommez le en Part1_Question_7.
- Créez les deux modules d'interfaçage en charge de la traduction du protocole ``fifo'' en protocole cycle près.
- Modifiez le module RGB_2_YCbCr afin de le rendre uniquement sensible à l'horloge (VHDL-like) et implanter le comportement décrit ci-dessus.
- Compilez le modèle puis lancez la simulation.
- Valider le fonctionnement de votre modèle.

Vous venez de valider fonctionnellement votre modèle. Cette validation vous a permis de valider la fonctionnalité du modèle, mais en aucun cas le comportement temporel des composants. Afin de valider le comportement temporel du système, le plus simple est d'étudier les signaux d'entrée et de sortie de votre module.

- Instrumentez votre modèle à l'aide des fonctions SystemC spécialisées dans l'extraction de chronogrammes (**sc_trace**, etc.).
- Compilez le modèle puis lancez la simulation.
- Ouvrez le fichier contenant les chronogrammes à l'aide de l'outil nommé **gtkwave**.

Chapitre

5

De SystemC à l'implantation VHDL

1. INTRODUCTION

2. EVALUATION ET EXPLOITATION DE LA DYNAMIQUE DES DONNEES ET DES CALCULS

Afin d'apporter une réponse globale à la gestion de la dynamique des données au sein des architectures générées, nous avons décidé de modifier l'ensemble du processus de synthèse d'architectures afin tout d'abord d'estimer la dynamique des calculs et de données avant d'optimiser la génération de l'architecture. L'étape d'estimation de la dynamique des données à partir des informations fournies par l'utilisateur est détaillée dans [REF]. Cette dernière permet à de réaliser une évaluation de la dynamique des données et des calcul à partir de la dynamique des entrées de l'algorithme et d'une bibliothèque spécifiant le comportement des opérations usuelles. La dynamique calculée par une approche basée sur la propagation de l'espace de variation pour chaque opération de la description algorithmique assure l'absence de débordement lors de l'exécution.

3. CONCEPTION DU CIRCUIT EN VHDL

Le modèle SystemC que vous avez décrit est très proche de la description VHDL RTL du composant. Dans cet TP nous supposerons que la finalité de l'étude est d'intégrer toute la chaîne de compression JPEG au sein d'un FPGA. Vous allez maintenant devoir écrire l'entité VHDL correspondant à la transformation RGB_2_YCbCr.

Remarque

> Vous ferez bien attention à écrire une entité VHDL correspondant au code SystemC c.f. l'entité décrite ci-dessous.

```
entity rgb2ycbcr_16b is
    port(
        clk     : in  STD_LOGIC;
        i_data  : in  STD_LOGIC_VECTOR(23 downto 0);
        o_data  : out STD_LOGIC_VECTOR(23 downto 0);
        i_valid : in  STD_LOGIC;
        o_valid : out STD_LOGIC
    );
end rgb2ycbcr_16b;
```

De plus vous veillerez à bien sélectionner les bons sous ensembles de bits pour vos données :

```
b <= i_data (23 downto 16);
v <= i_data (15 downto  8);
r <= i_data ( 7 downto  0);

o_data (23 downto 16) <= cr;
o_data (15 downto  8) <= cb;
o_data ( 7 downto  0) <= y;
```

A vous de jouer…

- Dessinez un schéma bloc du système que vous allez concevoir.
- Ecrivez le code VHDL.
- Validez le comportement de votre code VHDL à l'aide d'un testbench.

4. SIMULATION CONJOINTE AVEC MODELSIM

Le langage SystemC couplé avec l'utilisation de l'outil Modelsim vous permet de simuler des modèles contenant à la fois des modules décrits en SystemC et en VHDL. Vous allez exploiter cette capacité afin de valider de manière plus exhaustive le comportement de votre entité VHDL.

Cette possibilité de mixer des langages de modélisation et de description va vous permettre de continuer a vérifier votre composant VHDL de la même manière que précédemment. Ainsi on pourra vérifier son fonctionnement en

regardant l'image générée. Cette approche est bien plus efficace et agréable que celle consistant à observer des dizaines ou des centaines de signaux dans l'interface de Modelsim.

- ~~Ouvrez le projet Modelsim fourni par votre enseignant.~~
- ~~Validez le comportement de votre code VHDL à l'aide d'un testbench.~~
- Mettez votre fichier VHDL sur une clef USB et allez co-simuler votre système sur l’ordinateur de votre enseignant.

Co-simulation

Afin de compiler et de simuler vos modules SystemC en même temps que votre module VHDL, vous prendrez soin d’effectuer les actions suivantes :

> Click droit dans la fenêtre projet, puis « compile all »

> Dans la zone de commande :

> sccom -link

> vsim top

> run 100 ms

Chapitre 6

Validation fonctionnelle du matériel (VHDL)

Résumé : Ajouter de la reconfiguration dynamique dans les systèmes permet d'optimiser les coûts d'utilisation, d'améliorer les performances et potentiellement la consommation d'énergie. Malheureusement, tous ces bénéfices ne sont atteignables qu'au prix d'un effort significatif de conception. Il est donc important de concevoir des architectures efficaces, mais aussi les flots de conception permettant de manipuler ces architectures de manière simple et intuitive. Je présente dans ce chapitre mon activité autour de la conception d'architectures reconfigurables dynamique- ment et des méthodes associées. Le graphique suivant présente chronologiquement les différents projets ainsi que les étudiants de masters et les thèses impliqués dans cette thématique.

5. INTRODUCTION

5.1 Evaluation et exploitation de la dynamique des données et des calculs

Afin d'apporter une réponse globale à la gestion de la dynamique des données au sein des architectures générées, nous avons décidé de modifier l'ensemble du processus de synthèse d'architectures afin tout d'abord d'estimer la dynamique des calculs et de données avant d'optimiser la génération de l'architecture. L'étape d'estimation de la dynamque des données à partir des informations fournies par l'utilisateur est détaillée dans [REF]. Cette dernière permet à de réaliser une évaluation de la dynamique des données et des calculs à partir de la dynamique des entrées de l'algorihtme et d'une bibliothèque spécifiant le comportement des opérations usuelles. La dynamique calculée par une approche basée sur la propagation

de l’espace de variation pour chaque opération de la description algorithmique assure l’absence de débordement lors de l’execution.

Une fois l’étape d’analyse exécutée, le modèle interne est annoté afin d’indiquer la taille et le type (signé, non signé) des données et opérations. C’est à partir de ces informations que le processus de synthèse a été transformé afin d’améliorer la conception de l’architecture matérielle. Afin de (a) limiter la complexité calculatoire de l’étape de synthèse, (b) ne pas contraindre les étapes d’ordonnancement et d’assignation, nous avons décidé de réaliser une étape de sélection et d’allocation des ressources à base d’opérateurs « indéterminés » (sans gestion de la dynamique).

Chapitre 6

Simulation de la chaine JPEG intégrale

Résumé : Ajouter de la reconfiguration dynamique dans les systèmes permet d'optimiser les coûts d'utilisation, d'améliorer les performances et potentiellement la consommation d'énergie. Malheureusement, tous ces bénéfices ne sont atteignables qu'au prix d'un effort significatif de conception. Il est donc important de concevoir des architectures efficaces, mais aussi les flots de conception permettant de manipuler ces architectures de manière simple et intuitive. Je présente dans ce chapitre mon activité autour de la conception d'architectures reconfigurables dynamique- ment et des méthodes associées. Le graphique suivant présente chronologiquement les différents projets ainsi que les étudiants de masters et les thèses impliqués dans cette thématique.

6. SIMULATION GLOBALE DU SYSTEME

Dans cette dernière partie nous allons nous intéresser à l'application complète. Nous supposerons que ce travail est collaboratif (votre enseignant constituant le reste de l'équipe).

6.1 Evaluation et exploitation de la dynamique des données et des calculs

Afin d'apporter une réponse globale à la gestion de la dynamique des données au sein des architectures générées, nous avons décidé de modifier l'ensemble du processus de synthèse d'architectures afin tout d'abord d'estimer la dynamique des calculs et de données avant d'optimiser la génération de l'architecture. L'étape d'estimation de la dynamque des données à partir des informations fournies par l'utilisateur est détaillée dans [REF]. Cette dernière permet à de réaliser une évaluation de la dynamique des données et des calculs à partir de la dynamique des entrées de

l'algorihtme et d'une bibliothèque spécifiant le comportement des opérations usuelles. La dynamique calculée par une approche basée sur la propagation de l'espace de variation pour chaque opération de la description algorithmique assure l'absence de débordement lors de l'execution.

\part{Modélisation de la chaine JPEG complète}
%%%

\section*{Q1. Simulation globale du système}
%%%

L'ensemble des fichier nécessaires à cette partie du TP sont disponibles dans le repertoire (Part3_Question_1).
Ce répertoire contient tous les modules nécessaires à la modélisation de la chaîne. Toutefois l'ensemble des comportement n'a pas été décrit.

Avant de commencer à coder:

\begin{itemize}
\item Ouvrez le fichier \textit{main.cpp} et analysez les modules mis en jeu pour réaliser le modèle SystemC.
\item Dessinez sur feuille de papier le modèle SystemC (modules mis en oeuvre et interconnections).
\end{itemize}

Avant de pouvoir compiler et simuler la chaine de compression JPEG au niveau comportemental, vous allez devoir:

\begin{itemize}
\item Ouvrez le fichier nommé \textit{Conversion.cpp} et complétez le fichier à l'aide du code que vous avez développé précédemment (version flottante de l'algorithme).
\item Ouvrez le fichier nommé \textit{iConversion.cpp} et complétez le fichier à l'aide du code que vous avez développé précédemment (version flottante de l'algorithme).
\item Ouvrez le fichier nommé \textit{RLE.cpp} et complétez le fichier à l'aide du code que vous avez développé précédemment (Partie 2, Question 2).
\item Ouvrez le fichier nommé \textit{iRLE.cpp} et complétez le fichier à l'aide du code que vous avez développé précédemment (Partie 2, Question 2).
\item Ouvrez le fichier nommé \textit{Serializer.cpp} et complétez afin d'implanter le comportement attendu (décrit ci-dessous).
\end{itemize}

L'objectif du module \textit{Serializer} est de permettre l?interconnexion du module YCbCr avec le module DCT. Ces deux composants sont algorithmiquement incompatibles:

\begin{itemize}
	\item Le module YCbCr consomme un triplet $\{R,G,B\}$ et produit un triplet $\{Y,Cb,Cr\}$.
	\item Le module DCT consomme un 64 données $\{Y_0, \ldots Y_{63}\}$, produit 64 données $\{y_0, \ldots y_{63}\}$ avant de réaliser le même traitement sur les données de Cr et Cb.
\end{itemize}

Afin de permettre l'échange de données entre ces 2 composants, vous allez devoir concevoir un \textit{wrapper}. Ce dernier va revoir en entrée $\{Y_0, Cr_0, Cb_0,\ldots Y_{63},Cr_{63},Cb_{63}\}$ et il devra produire en sortie $\{Y_0 \ldots Y_{63}, Cb_0 \ldots Cb_{63}, Cr_{0} \ldots Cr_{63}\}$.

Une fois que le comportement de tout les composants a été décrit, vous pouvez simuler le modèle. Pour cela

\textbf{Remarque:} Une fois que votre modèle est fonctionnel, mesurer le temps d'exécution nécessaire à la compression de l'image de test. Pour cela, vous utiliserez la commande $time$ disponible dans le terminal.

\section*{Q2. Modification des interfaces de communication (bit-accurate)}
%%

Comme vous l'avez sûrement remarqué lorsque vous avez dessiné le modèle dans la question précédente, toutes les interfaces de communication du modèle manipulent des données de type int. Comme vous l'avez aussi vu précédemment, le fait d'utiliser un type de donné non adapté peut masquer des problèmes de conception.

Vous allez donc modifier votre système afin de faire en sorte que les interfaces de communication soient définies précisément. Pour cela, vous vous appuierez sur les informations suivantes:

\begin{itemize}
	\item L'entrée et la sortie du module $RGB \rightarrow YCbCr$ manipulent des données non signées, codées sur 8 bits.
	\item L'entrée du module DCT manipule des données non signées, codées sur 8 bits. Sa sortie transmet des informations signées codées sur 12 bits.

\item L'entrée et la sortie du module de $Quantification$ manipulent des données signées, codées sur 12 bits.
\item L'entrée et la sortie du module de RLE manipulent des données signées, codées sur 12 bits (pour le moment).
\end{itemize}

A l'aide de ces informations, modifiez tout d'abord sur papier le format des interconnections entre les modules SystemC. Afin de coder spécifier efficacement les entrées et les sorties des modules, il vous est demandé d'utiliser les classes $sc_uint<N>$ et $sc_int<N>$.

\begin{itemize}
\item Dupliquez le répertoire de lié à la question précédente et renommez le en Part3_Question_2.
\item Répercutez les modifications dans le code SystemC.
\item Compilez et simulez votre modèle.
\item Vérifiez que les résultats sont strictement identiques.
\end{itemize}

\section*{Q3. Intégration de vos modèles SystemC bit-accurate}
%%
%%%%%%%%%%%%%%%%%%%%%%%%

Maintenant que votre chaîne est fonctionnelle et que vos interface sont correctement dimensionnées, vous allez pouvoir réutiliser les modèles développer précédemment:

\begin{itemize}
\item Dupliquez le répertoire de lié à la question précédente et renommez le en Part3_Question_3.
\item Insérez dans votre modèle les modèles bit-près des composants RLE \& RGB_2_YCbCr.
\item Compilez et simulez votre modèle.
\item Vérifiez que les résultats sont strictement identiques.
\end{itemize}

\section*{Q4. Etude des performances temporelles}
%%
%%%%%%%

Nous allons maintenant mesurer le temps nécessaire à la compression d'une image JPEG. Pour cela vous allez devoir intégrer les contraintes temporelles suivantes :

\begin{itemize}

\item Le traitement d'un triplet de données par le bloc RGB_2_YCbCr est de $30ns$.
\item Votre adaptateur ($RGB_2_YCbCr \rightarrow DCT$) nécessite $10ns$ pour lire un jeu de données en entrée et les stocker en mémoire. La restitution des données se fait à hauteur d'une donnée toutes les $10ns$ quand tous les blocs ont été reçus. Les deux étapes ne sont pas recouvrantes.
\item Le composant DCT consomme une donnée toutes les $10ns$ et les produit au même rythme. La latence du composant est de $800ns$ (délai séparant la première entrée de la première sortie).
\item Le Zig-Zag consomme une donnée toutes les 10ns et les restitue à la même vitesse lorsque toutes les données ont été reçues.
\item Le module de quantification consomme une donnée, la traite et restitue le résultat $10ns$ plus tard. Il consomme immédiatement la donnée suivante.
\item Le composant RLE est apte à consommer une donnée toutes les $10ns$ et les restitue lorsque cela est nécessaire avec un temps de traitement de $10ns$.
\item
\end{itemize}

L'ensemble de ces informations vous a été fourni par les autres concepteurs travaillant sur le projet. Vote objectif est d'estimer les performances du système final (lorsque tous les blocs matériels auront été assemblés).

\begin{itemize}
\item Modifier les modèles SystemC afin d'intégrer ces contraintes temporelles,
\item Compiler et simuler votre système et mesure le temps nécessaire à la compression de l'image.
\item Seriez vous capable (si jamais vous implantez tout le système) de tenir un débit de 24 images par seconde ?
\end{itemize}

\section*{Q5. Simulation multi-niveau}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Maintenant que votre modèle est réaliste (temporellement), vous allez pouvoir commencer a intégrer les développements VHDL que vous avez précédemment conçus.

\section*{Q6. Co-Simulation: parties logicielles \& matérielles}
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%

Le dernier point que nous aborderons dans ce projet consiste à simuler le comportement de la chaîne de compression complète à l'aide de votre description SystemC plus de partie déportées sur FPGA. En effet dans cette partie nous allons porter sur carte la description VHDL du module RGB_2_YCbCr afin de réaliser une co-simulation (logicielle \& matérielle).

\begin{itemize}
    \item Dupliquez le répertoire contenant la dernière version de votre modèle SystemC.
    \item Remplacez le code SystemC du module RGB_2_YCbCr par le code fourni par votre enseignant.
    \item Voyez avec votre enseignant la procédure à mettre en oeuvre afin de porter votre code VHDL sur carte.
    \item Simulez la chaîne de compression JPEG.
\end{itemize}

Chapitre 6

Synthèse HLS des modules SystemC

Résumé : L'objectif de cette partie est de vous permettre d'utiliser les connaissances acquises durant le cours dédié à la synthèse de haut niveau (HLS). Dans le cadre de ce TP nous utiliserons un outil industriel développé par Xilinx pour générer des architectures matérielles sous contraintes.

Note : Pour toute question spécifique à l'outil « Vivado HLS » et/ou au langage C, C++ ou SystemC supporté par ce dernier, référez vous au document UG902 fourni par Xilinx.

1. INTRODUCTION

Lors des TPs précédents vous avez raffiné progressivement votre algorithme afin de le passer d'une description comportementale de type Matlab à une description matérielle de niveau RTL. Les dernières transformations que vous avez réalisées ont été particulièrement complexes et chronophages (le passage des modèles au niveau CABA).

Pour accélérer l'implantation des modèles algorithmiques décrits en C/C++, C++ ou SystemC des outils de HLS ont été développés. Ces derniers conçoivent à votre place l'architecture matérielle qui implémente l'algorithme (instanciation des ressources de calcul, des mémoires RAM, des registres et conception du séquenceur).

Afin de mieux appréhender ce type de méthodologie, vous allez dans un premier temps implanter une application « foo ». Dans un second temps vous vous ferez vos armes avec votre module de conversion colorimétrique « RGB2YUV ». Puis en fonction du temps restant vous essayerez d'implanter efficacement les différents modules de la chaine de compression JPEG.

2. PRISE EN MAIN DE L'OUTIL VIVADO-HLS

La première partie du TP vise à vous familiariser avec l'outil Vivado de Xilinx. Vous implanterez votre premier circuit RTL décrit en langage C/C++ et étudierez l'impact des pragma supportés par l'outil sur les architectures générées. Dans un premier temps, réalisez les étapes suivantes :

- Lancez l'outil « Vivado HLS 2014.3 » via son raccourci sous Windows.
- Créez un nouveau projet « File : New project ». Nommez ce premier projet « test_tp_3a_se ». N'ajoutez pas de fichiers dans le projet pour le moment. Dans le dernier onglet lorsque vous devez indiquer la cible, utiliser le sous-menu « Boards » puis « ZedBoard… ». Valider et créez le projet vide.
- Maintenant créez et ajouter deux fichiers dans votre projets. Pour cela cliquez sur « source » dans le panel de gauche et faites « new file ». Créez un fichier nommé « my_module.cpp » et un autre nommé « my_module.h ». Vous prendrez soin de ranger ces fichiers dans un répertoire nommé « src ».
- Cliquez sur votre projet à l'aide du bouton droit de la souris (dans le panneau de gauche) et sélectionnez « project settings » Dans le panneau nommé « synthesis », indiquez que le nom du module SystemC à implanter en VHDL est « my_module ».
- Décrivez en SystemC un module nommé « my_module ». Ce module sera compose d'une entrée "x" de type sc_fifo, une sortie "y" de type sc_fifo et d'une entrée de type sc_in<bool> pour l'horloge "clk".
- Implantez le comportement du module (Y = 1000 / (2 * (X * X) – 7 * X + 1)) à l'aide d'un SC_CTHREAD sensible au front montant de l'horloge. Vous prendrez soin de déclarer « x » et « y » comme des données entières (int).
- Appuyez sur le bouton « Run C synthesis » afin de vérifier que votre code SystemC est syntaxiquement correct.
- Avant de générer l'architecture VHDL implantant le comportement décrit, vous allez vérifier le bon fonctionnement du code logiciel que vous avez écrit. Pour cela, récupérez les fichiers qui décrivant un testbench ici. Ajouter ces fichiers dans votre projet dans la section testbench. Enfin, lancez la simulation de votre système et valider les valeurs obtenues en sortie.
- Maintenant que vous avez démontré/observé que le comportement de votre module SystemC est correct, réalisez sa synthèse. Pour cela cliquez, sur la petite flèche verte dans la barre d'outil.

- Une fois l'opération terminée, une nouvelle fenêtre est apparue. Analysez les informations fournies par l'outil après synthèse. Une vue plus détaillée des informations post-synthèse est disponible si vous cliquez sur la perspective « Analysis » en haut à droite de l'outil. Activez cette perspective et analysez toutes les informations fournies (performances, ressources).
- Vous venez de générer une architecture matérielle de niveau RTL (sans vous en rendre compte). Les fichiers VHDL, Verilog et SystemC correspondant sont disponibles dans le panneau de gauche dans la sous rubrique « solution 1 ». Ouvrez les fichiers VHDL et observez les.
- Reconstituez le circuit conçu pour vous par l'outil Vivado.

A partir du rapport de synthèse (ordonnancement des opérations dans le temps) + des fichiers VHDL essayer de dessiner le circuit que l'outil a conçu pour vous.

- Les caractéristiques matérielles que vous avez analysées précédemment sont des estimations faites par l'outil Vivado. Afin de connaître de manière précise les performances du système, cliquez sur le menu « Solution » puis sur « Export RTL ». Sélectionnez « Evaluate » et choisissez le mode « VHDL » puis validez. Comparez les performances fournies dans ce nouveau rapport avec les estimations fournies précédemment par l'outil.
- Vous venez d'analyser les performances du circuit, mais vous avez oubliez de vérifier son fonctionnement… Corriger cela en lançant une simulation du circuit en mode SystemC CABA (Cycle Accurate, Bit Accurate).

PARTIE A MODIFIER POUR QU'ILS REUTILISENT MON CODE ! LEUR FAIRE REMARQUE QUE LE MODULE REQUIRT UN TEMPS DE CALCUL ! **Plus possible avec la nouvelle version** ?

- Afin de rendre la simulation possible, remplacer l'instanciation de « my_module » par l'instanciation de « my_module_rtl_wrapper » dans votre testbench.
- Transformez votre programme source afin de faire en sorte que les entrées « x » et la sortie « y » soient des codées sur 16 bits (tout comme les calculs intermédiaires). Recommencez toutes les étapes et comparez les performances des deux systèmes.

Pour vous simplifier la vie, nous vous conseillons d'utiliser : la génération de solutions multiples & la comparaison de design

- Transformez à nouveau votre programme source afin manipuler des données de type « int ». Modifiez la fréquence d'horloge ciblée post-synthèse. Essayez avec les périodes 5 ns, 10 ns, 20 ns et 50 ns.
- Observer l'impact de ces modifications sur les résultats de synthèse et calculez les performances en terme de débit et de latence pour chacun des circuits générés. Mettez ces chiffres en regard avec la complexité matérielle des circuits.
- Maintenant vous allez changer la cible matérielle indiquée dans les paramètres de synthèse. Choisissez d'abord le FPGA présent sur la carte d'évaluation « Virtex-7 VC707 ». Faites la synthèse de votre description et interprétez les résultats.
- Faites de même lorsque la cible est un Kintex-7 (carte KC705).

Note

> Afin faciliter la comparaison des performances des modules et l'interprétation des résultats vous spécifierez une contrainte de 10 ns pour la fréquence de fonctionnement.

3. IMPLANTATION DU MODULE RGB_2_YUV

Maintenant que vous maitrisez les concepts de base de l'outil Vivado HLS, vous allez mettre en œuvre vos connaissances afin d'intégrer votre module RGB_2_YUV développé en durant les séances de TPs précédentes. Dans cette partie, vous focaliserez votre attention sur les performances obtenues en fonction du niveau de description du modèle SystemC.

- Reprenez le modèle SystemC réalisant la conversion RGB_2_YUV à l'aide d'opérations flottantes.
- Adaptez cette description afin de la rendre synthétisable. Procédez ensuite à la synthèse du modèle à l'aide de l'outil Vivado HLS.
- Analyser les résultats de synthèse.
- Procéder de la même manière avec le modèle SystemC qui réalise la conversion RGB_2_YUV à l'aide de données de type sc_fixed.
- Faites la synthèse de du modèle à l'aide de l'outil Vivado HLS. Réitérez pour différents formats de codage (nombre de bits pour la partie fractionnaire).
- Analyser les résultats de synthèse.

- Faites de même avec le modèle SystemC réalisant la conversion RGB_2_YUV à l'aide de données de type sc_int.
- Faites la synthèse de du modèle à l'aide de l'outil Vivado HLS. Réitérez pour différents formats de codage (nombre de bits pour la partie fractionnaire).
- Analysez les résultats de synthèse.
- Comparez les performances obtenues vis-à-vis du module VHDL que vous avec décrit à la main durant les TP précédents.
- Comparez les performances obtenues vis à vis du module VHDL que vous avec décrit à la main durant les TP précédents.
- Afin d'améliorer les performances, il est possible de travailler sur plusieurs données en parallèle (données provenant d'itérations successives). Pour cela, étudiez les pragmas [HLS STREAM] et [HLS PIPELINE] présentés dans la documentation de l'outil Vivado HLS.
- Améliorez les performances de votre module en prenant en considération le fait que 64 échantillons sont toujours traités.

4. APPLICATION A UN CAS D'ETUDE REEL

Apres ces quelques exemples pédagogiques, vous avez suffisamment de connaissances pour essayer d'intégrer la chaine de compression JPEG : RGB_2_YUV + DCT2d + Quantification + ZigZag.

- Créez un projet par module pour commencer.
- Faites la synthèse de chacun des modules SystemC.
- En fonction des performances obtenues, si cela est nécessaire, optimisez la description SystemC et/ou ajoutez des pragma.
- Une fois que l'ensemble des modules ont été synthétisé, développez un modèle SystemC qui regroupe l'ensemble de la chaine de traitement. Vous raccorderez les différents modules à l'aide de sc_fifo de profondeur 64.
- Faites la synthèse de la chaine JPEG. Quelles performances obtenez vous ?
- Combien de temps, faut il pour traiter une image de résolution 4k @ 100MHz ?
- Optimisez votre description SystemC pour obtenir de meilleures performances.

Chapitre 6

Test et validation des modules

Résumé : Ajouter de la reconfiguration dynamique dans les systèmes permet d'optimiser les coûts d'utilisation, d'améliorer les performances et potentiellement la consommation d'énergie. Malheureusement, tous ces bénéfices ne sont atteignables qu'au prix d'un effort significatif de conception. Il est donc important de concevoir des architectures efficaces, mais aussi les flots de conception permettant de manipuler ces architectures de manière simple et intuitive. Je présente dans ce chapitre mon activité autour de la conception d'architectures reconfigurables dynamique- ment et des méthodes associées. Le graphique suivant présente chronologiquement les différents projets ainsi que les étudiants de masters et les thèses impliqués dans cette thématique.

5. TEST ET VERIFICATION DES PARTIES LOGICIELLES

6. TEST ET VERIFICATION DES PARTIES MATERIELLES

6.1 Evaluation et exploitation de la dynamique des données et des calculs

Afin d'apporter une réponse globale à la gestion de la dynamique des données au sein des architectures générées, nous avons décidé de modifier l'ensemble du processus de synthèse d'architectures afin tout d'abord d'estimer la dynamique des calculs et de données avant d'optimiser la génération de l'architecture. L'étape d'estimation de la dynamique des données à partir des informations fournies par l'utilisateur est détaillée dans [REF]. Cette dernière permet à de réaliser une évaluation de la dynamique des données et des calculs à partir de la dynamique des entrées de

l'algorihtme et d'une bibliothèque spécifiant le comportement des opérations usuelles. La dynamique calculée par une approche basée sur la propagation de l'espace de variation pour chaque opération de la description algorithmique assure l'absence de débordement lors de l'execution.

Une fois l'étape d'analyse exécutée, le modèle interne est annoté afin d'indiquer la taille et le type (signé, non signé) des données et opérations. C'est à partir de ces informations que le processus de synthèse a été transformé afin d'améliorer la conception de l'architecture matérielle. Afin de (a) limiter la complexité calculatoire de l'étape de synthèse, (b) ne pas contraindre les étapes d'ordonnancement et d'assignation, nous avons décidé de réaliser une étape de sélection et d'allocation des ressources à base d'opérateurs « indéterminés » (sans gestion de la dynamique).

Chapitre

6 Intégration finale sur plateforme Zinq

Apres tous les éfforts consentits jusuq'à présent, nous allons implanter l'ensemble des composants logiciels et matériels dans notre système. Cette dernière étape vous permetttra de mesurer les gains obtenus en terme de débit et latence.

7. INTRODUCTION

7.1 Evaluation et exploitation de la dynamique des données et des calculs

Afin d'apporter une réponse globale à la gestion de la dynamique des données au sein des architectures générées, nous avons décidé de modifier l'ensemble du processus de synthèse d'architectures afin tout d'abord d'estimer la dynamique des calculs et de données avant d'optimiser la génération de l'architecture. L'étape d'estimation de la dynamque des données à partir des informations fournies par l'utilisateur est détaillée dans [REF]. Cette dernière permet à de réaliser une évaluation de la dynamique des données et des caluc l à partir de la dynamique des entrées de l'algorihtme et d'une bibliothèque spécifiant le comportement des opérations usuelles. La dynamique calculée par une approche basée sur la propagation de l'espace de variation pour chaque opération de la description algorithmique assure l'absence de débordement lors de l'execution.

Une fois l'étape d'analyse exécutée, le modèle interne est annoté afin d'indiquer la taille et le type (signé, non signé) des données et opérations. C'est à partir de ces informations que le processus de synthèse a été transformé afin d'améliorer la conception de l'architecture matérielle. Afin de (a) limiter la complexité calculatoire de l'étape de synthèse, (b) ne pas contraindre les étapes d'ordonnancement et d'assignation, nous avons décidé

de réaliser une étape de sélection et d’allocation des ressources à base d’opérateurs « indéterminés » (sans gestion de la dynamique).