Boîte à outils pour la conception d'accélérateurs de traitement d'image

Stéphane Mancini

24 septembre 2025

Table des matières

Chapitre 1

Démonstration CatapultC

1.1 Démonstration CatapultC

La démonstration est celle du filtre FIR (Finite Impule Response) de MentorGraphics. La démonstration de base est dans le répertoire :
`/softslin/catapultc10_1b/Mgc_home/shared/examples/designs/filter_architectures/cxx/`
Cette démonstration est copiée dans l'archive : `Catapult/CC_demo/Demo_FIR`. Le fichier `FIR_FILTER_WALKTHROUGH_CPP.pdf`, copié dans ce répertoire, peut être consulté pour information.

Le répertoire `Demo` contient la version de la démonstration présentée en cours. Dans ce répertoire, vous trouverez les fichiers :

- `directives.tcl`, Pour lancer la HLS sur le projet.
  Dans `CatapultC`, le script s'exécute depuis le menu *'File -> run script -> choisir directives.tcl'*
- `simu_batch.tcl`, Pour lancer la simulation post-HLS.
  Idem, avec le fichier `simu_batch.tcl`
- `directives_sim.tcl`, Pour faire la HLS, suivie automatiquement de la simulation.

Le fichier

1.1.1 Fichiers d'entrée

Les fichiers C/C++ utilisés pour la HLS sont sélectionnés par le menu *'Task Bar -> Input File'*. Sélectionner les fichiers sources et indiquez ceux utilisés exclusivement pour la simulation, dont on ne fait pas la HLS, en marquant *Exclude* dans la case appropriée.

1.1.2 Compilation pré-HLS

Le menu *'Task Bar -> Hierarchy'* lance la compilation interne à CatapultC. Vérifiez les erreurs et modifiez le code C/C++ jusqu'à passer cette étape.

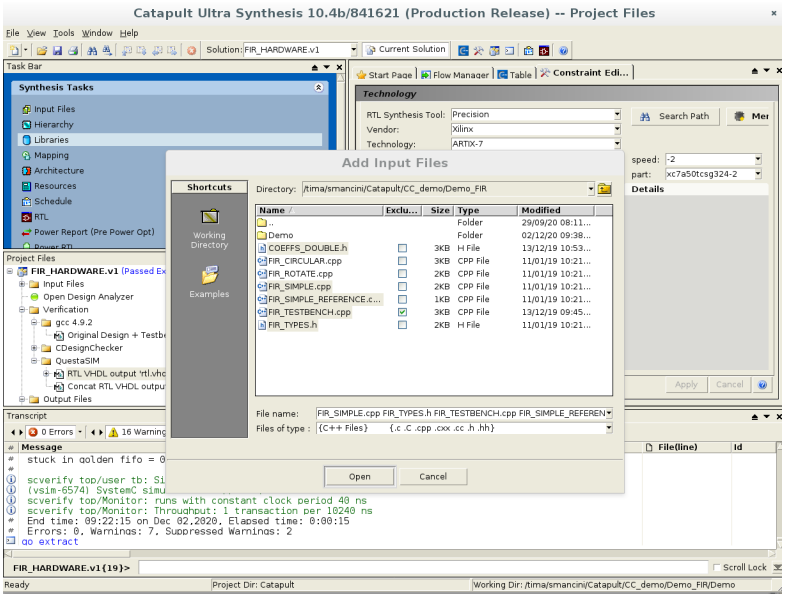


FIGURE 1.1 – Fichiers source

A cette étape vérifiez que la hiérarchie du projet est correcte que votre fonction C/C++ est bien le plus haut niveau (top).

1.1.3 Sélection de la technologie cible

Le menu *'Task Bar -> Libraries'* permet de cibler la technologie cible et le logiciel de synthèse logique post-HLS.

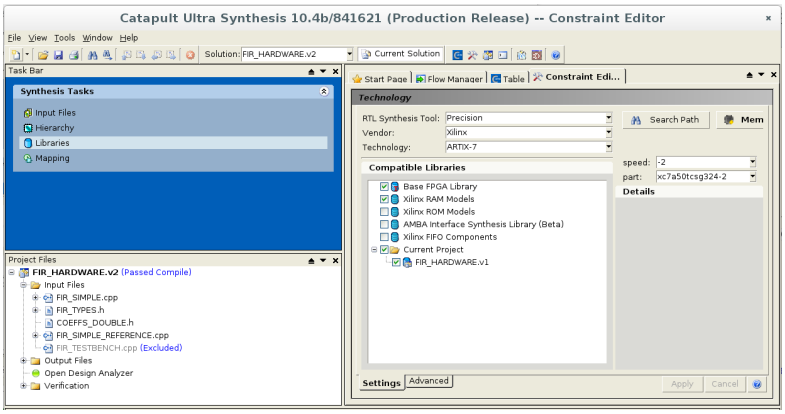
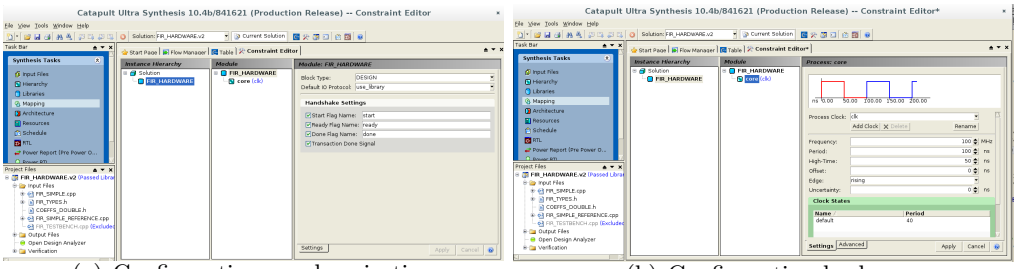


FIGURE 1.2 – Configuration de la technologie cible

1.1.4 Configuration des interfaces génériques

Le menu *'Task Bar -> Mapping'* configure les interfaces basiques du projet. La



(a) Configuration synchronisation (b) Configuration horloge

FIGURE 1.3 – Configuration des interfaces basiques

configuration de la synchronisation permer d'ajouter des signaux de contrôle du bloc matériel généré.

1.1.5 Configuration de l'architecture

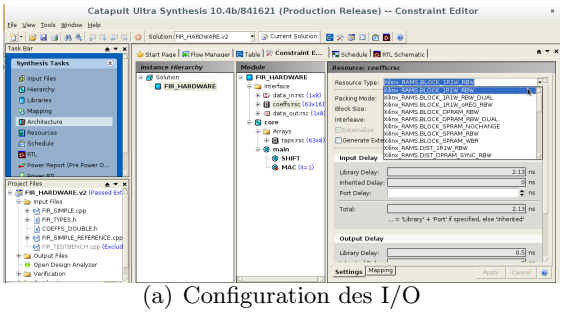
C'est la partie la plus importante, celle qui permettra d'optimiser l'architecture selon vos objectifs de performance. Le menu *'Task Bar -> Architecture'* vous permet de configurer

- La technologie pour implémenter des interfaces et comment transformer les tableaux (fils, registres, mémoires simple/double port)
- La technologie pour implémenter les tableaux (registres, mémoire simple/double port)
- La façon de transformer les boucles
  - Déroulage de boucle (total/partiel), pour paralléliser ou non les itérations quand les dépendances le permettent
  - Implémenter un pipeline pour optimiser le débit et la latence des calculs

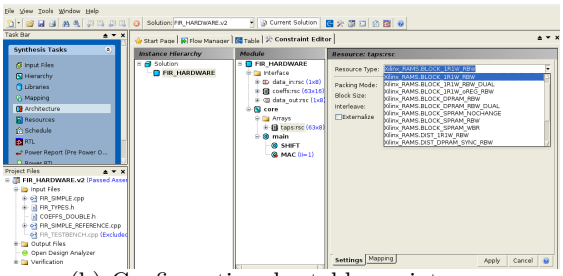
1.1.6 HLS et analyse

Les items suivants (*'Task Bar -> Resources'*, *'Task Bar -> Schedule*, *'Task Bar -> RTL*), permettent de décomposer la HLS en différentes étapes. Le chronogramme est disponible après le menu, et peut être analysé dans l'onglet *Schedule*, ou par l'icône *Gantt Chart* de la barre d'icône du dessus.

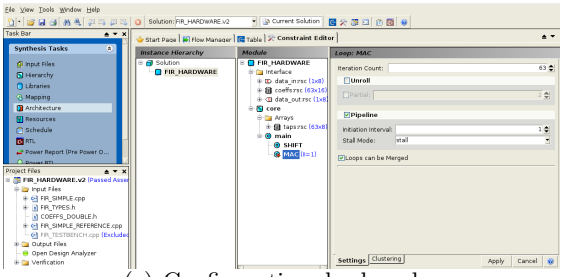
L'item *'Task Bar -> RTL* génère le VHDL et produit une schématique observable par l'icône *RTL Schematic* de la barre d'icône du dessus.



(a) Configuration des I/O



(b) Configuration des tableaux internes



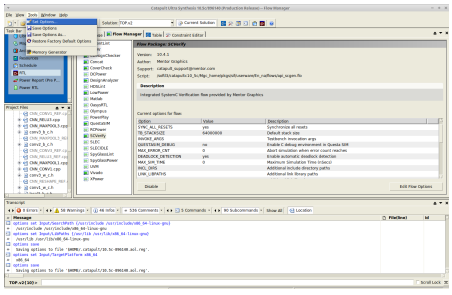
(c) Configuration des boucles

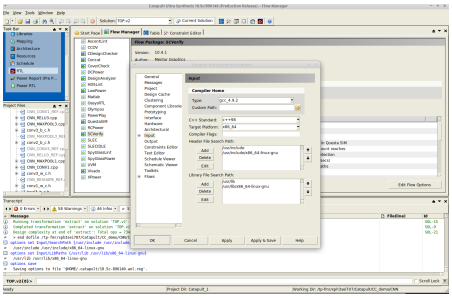
FIGURE 1.4 – Configuration de l'architecture

1.1.7 Simulation

La *simulation* du VHDL produit par CatapultC peut se lancer depuis l'interface graphique. La simulation VHDL se fait avec le bench écrit en C++ et CatapultC se charge de faire le lien entre le C++ et le VHDL.

Préalablement à la simulation, CatapultC doit être configuré de la façon suivante :





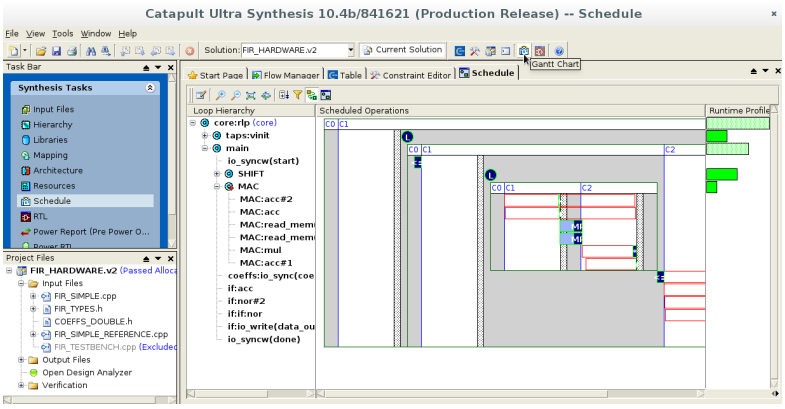


FIGURE 1.5 – Analyse du séquencement

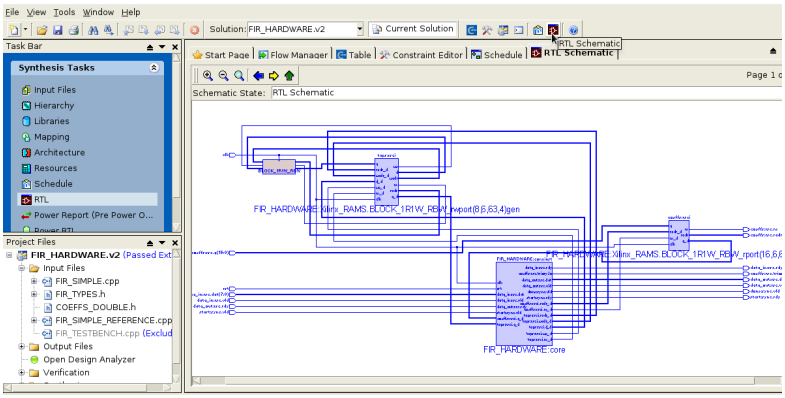
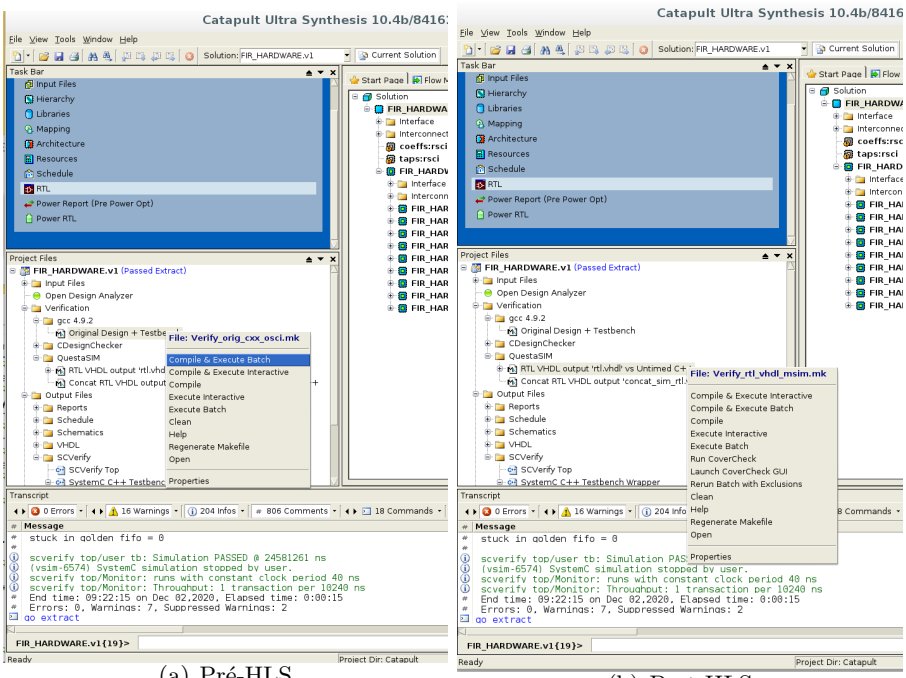


FIGURE 1.6 – Analyse du circuit produit

Une fois la configuration faite, la simulation est lancée de la façon suivante :

— Simulation pré-HLS, en fait la compilation et exécution du C++ :
'Project File -> Verification -> gcc 4.9.3 -> Original Design + TestBench (bouton droit) Compile & Execute Batch'

— Simulation post-HLS, en fait la compilation C++ du testbech et exécution du C++ + simulation VHDL généré :
'Project File -> Verification -> QuestaSIM -> RTH VHDL output ...(bouton droit) Compile & Execute Batch /ou/ Compile & Execute Interactive '
La version *Interactive* ouvre la fenêtre du simulateur VHDL, et montre les chronogramme. Le texte en provenance du C (printf, ...) est affiché dans la fenêtre de log de modelsim.
La version *Batch* ne montre pas les chronogrammes, le texte en provenance du C est affiché dans la fenêtre de log de CatapultC.
L'environnement de simulation compare automatiquement le résultat du VHDL et celui du C/C++ d'origine, et produit un rapport de comparaison affiché dans la fenêtre de log.



(a) Pré-HLS (b) Post-HLS

FIGURE 1.7 – Simulation pré et post HLS

Après la simulation, les fichiers lus et écrits par le code de simulation se trouvent dans le répertoire du projet. Au besoin cherchez les avec la commande bash `find`.

Chapitre 2

Méthodologie

2.1 Vue générale

La méthode de conception d'accélérateurs est une sous-partie de la méthode de conception de systèmes logiciels et matériels. On fera l'hypothèse que le partionnement logiciel/matériel a déjà été fait et on ne s'attachera qu'à la conception d'un accélérateur donné. Les étapes de conception décrites dans ce document sont des frontières 'théoriques', et, dans la réalité, on peut être amené à réaliser plusieurs étapes simultanément. L'intérêt de ce découpage est surtout de permettre une mise en perspective générale du projet et de se situer dans un flot de développement complexe.

Chaque étape est une spécialité en soi et l'optimisation globale d'un projet est assez difficile. Comme il est assez improbable de trouver la "meilleure" solution du premier coup (si tant est qu'il en exite une), dans un premier temps, l'objectif est de réaliser toutes les étapes le plus rapidement possible puis d'analyser les résultats pour ensuite optimiser le système.

Les grandes étapes de la conception de l'accélérateur sont les suivantes

— **R**éférence algorithmique, figure 2.1
  La référence algorithmique provient souvent d'une implémentation des équations mathématiques. De façon à s'affranchir de détails d'implantation, elle est le plus souvent en langage de 'haut niveau', comme matlab ou Python. Des langages commes C ou C++ sont possibles mais pas toujours souhaitables.

— **I**mplémentation virgule fixe, figure 2.2
  L'artimétique virgule fixe est mise en place. Chaque valeur est représentée soit par un nombre en virgule fixe soit par un entier. Dans un premier temps, un réglage grossier des précisions et dynamiques est suffisant pour valider l'implémentation.

— **I**mplémentation pour la HLS, figure 2.3

  En plus de la virgule fixe, les constructions algorithmiques tiennent compte des contraintes de la HLS :
  — Boucles bornées à bornes statiques
  — Allocation mémoire statique, c'est à dire sans allocation mémoire dynamique

(création d'objet, allocation mémoire explicite). Ceci nécessite une identification de toutes les mémoires et le calcul de leurs tailles, qui sera basé sur le pire cas.
- Détermination de toutes les constantes qui seraient des paramètres du code haut niveau (taille des tableaux, etc ...). Ces constantes peuvent être placées dans des fichiers de configuration sous forme de macros.
- Interactions avec l'environnement
  - Accès aux mémoires et contraintes associées (simple ou double ports)
  - Passage de paramètres par variables/registres ou mémoires
  - Synchronisation avec des flux de données
- **H**LS et réglage de l'architecture

  - Premiers tests de HLS
  - Analyse de performance et comparaison avec les performances prévues
  - Réglage des principaux paramètres architecturaux pour régler le compromis performance/surface : type de mémoire de chaque tableau interne/externe (SRAM simple ou double port, DFF), gestion des boucles (déroulement, pipeline), virgule fixe
- **V**érification fonctionnelle post-HLS, figure 2.4

  Le banc de test permet de vérifier que la HLS produit une architecture qui préserve la fonctionnalité.
- Synthèse logique
  Transformation du résultat de la HLS (RTL) en netlist
- **V**érification post synthèse logique, figures 2.6 2.5 2.7

  Théoriquement sous forme de simulation mais il est de temps en temps impossible de simuler le système et, dans ce cas, une émulation FPGA fait office de validation.
  Sur FPGA, l'accélérateur est connecté à son environnement (RAM, registres, FIFO) de façon à fournir des valeurs et stocker des résultats. Pour comparaison, il est également possible d'utiliser des mémoires de résultats produits aux étapes précédentes de façon à vérifier in-situ les résultats produits par l'accélérateur.

2.1.1 Conception d'accélérateur de traitement d'image

Concernant les accélérateurs de traitement d'image, la méthode mise en oeuvre est la suivante :

Référence algorithmique

Codée en Python. Les images d'entrée et sorties sont dans des fichiers, au format PGM. Les paramètres sont dans une donnée de type dictionnaire, facile à stocker dans un fichier

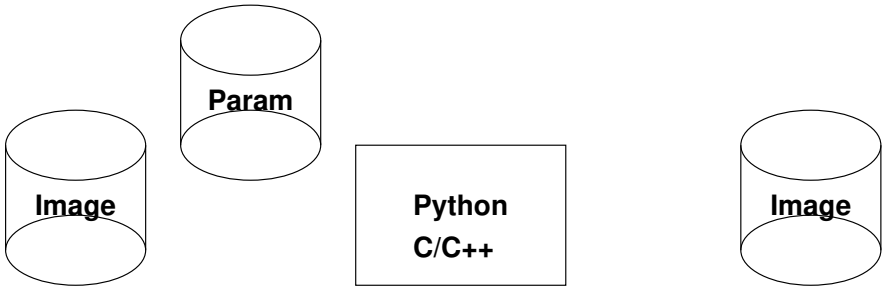


FIGURE 2.1 – Etape 1 : référence algorithmique

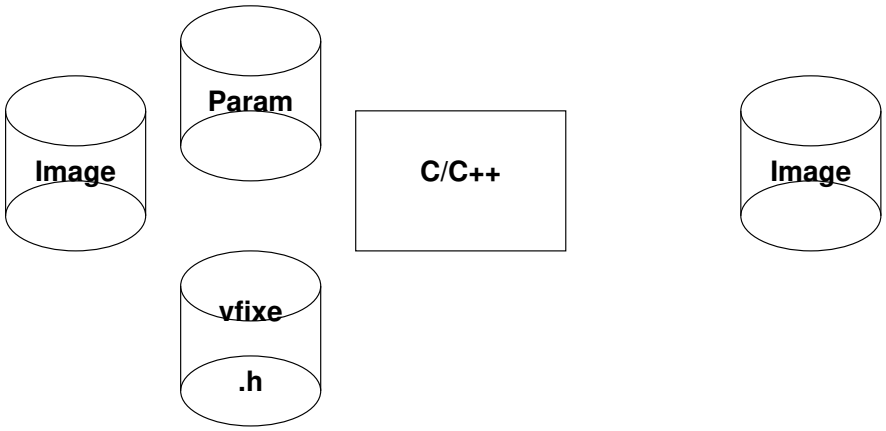


FIGURE 2.2 – Etape 2 : algorithme virgule fixe

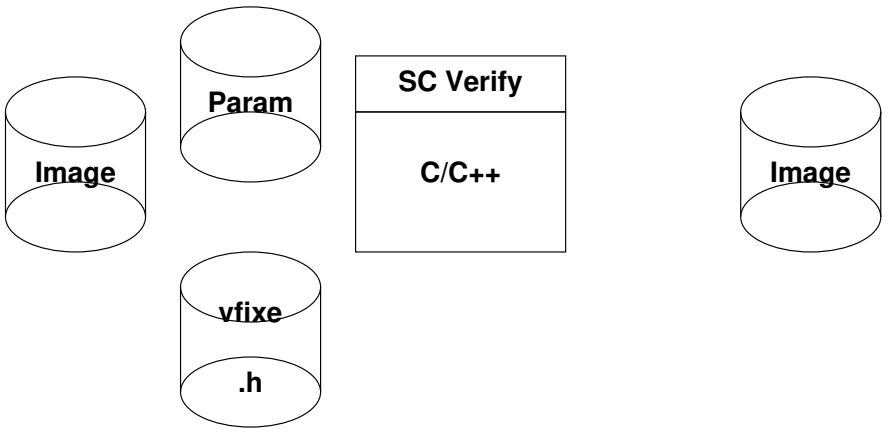


FIGURE 2.3 – Etape 3 : virgule fixe *pré*-HLS dans l'environnement SC-Verify

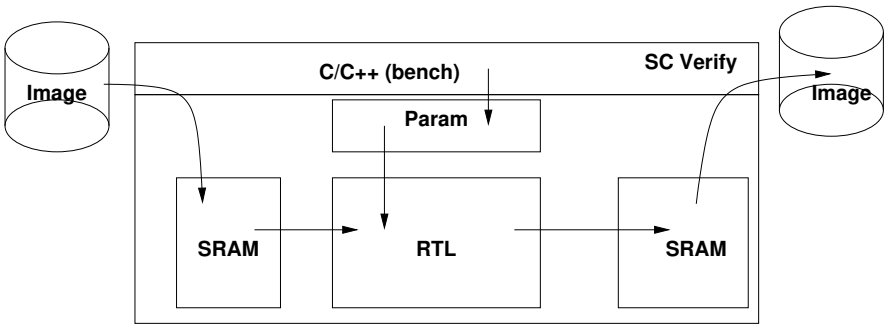


FIGURE 2.4 – Etape 4 : vérification *post*-HLS dans l'environnement SC-Verify

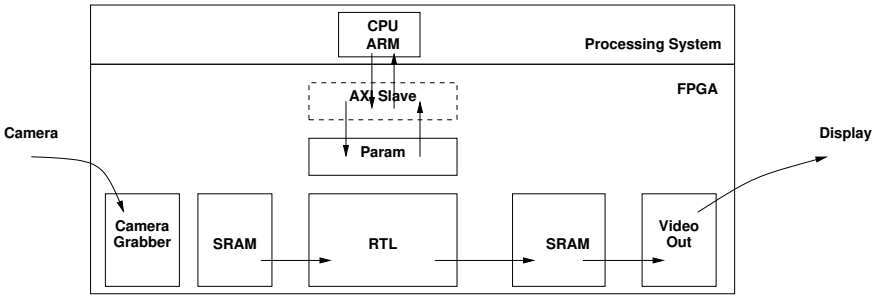


FIGURE 2.5 – Etape 5 : vérification par émulation ; Les données sont statiques (SRAM initialisée), les paramètres peuvent être statiques ou proviennent d'un SW par le bus AXI, les résultats sont affichés sur écran.

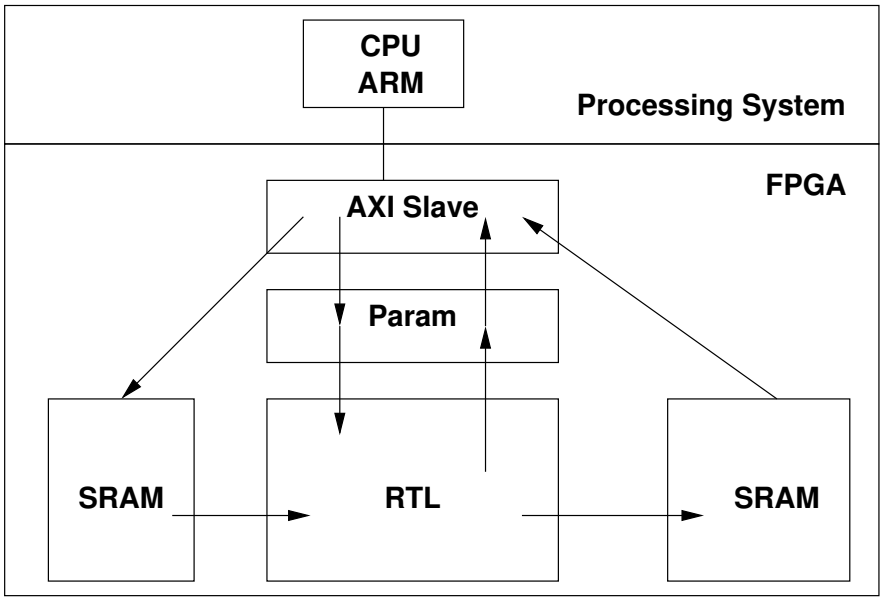


FIGURE 2.6 – Etape 6 : vérification par émulation dans un système HW/SW ; Les paramètres et les données viennent d'un logiciel (bare-metal par exemple), les résultats sont lus depuis le SW pour vérification.

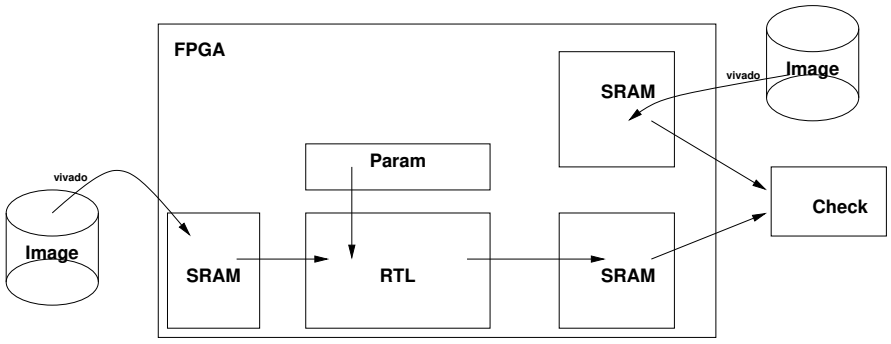


FIGURE 2.7 – Etape 6 bis : vérification par émulation dans un système HW/SW ; Les paramètres et les données sont statiques, les résultats sont comparés à des résultats attendus.

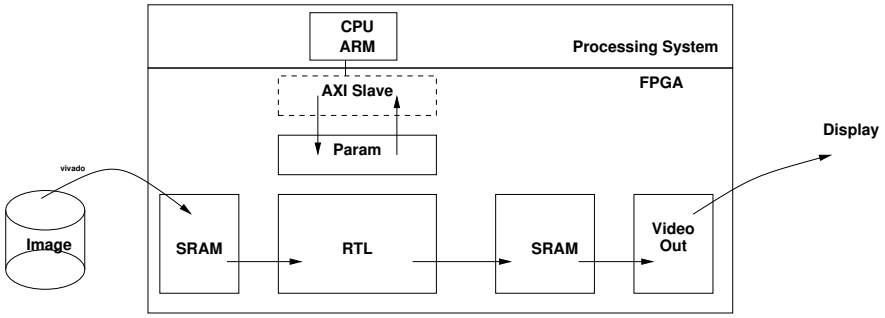


FIGURE 2.8 – Etape 8 : vérification par émulation ; Les données proviennent d'une caméra, les paramètres peuvent être statiques ou proviennent d'un SW par le bus AXI, les résultats sont affichés sur écran.

Implémentation virgule fixe et pré-HLS

Le code pour la HLS utilise la virgule fixe avec les type "AC types" de Mentor graphics [1]. L'algorithme est vérifié par l'exécution du code compilé, sur un PC, sans HLS. Les paramètres de virgule fixe sont sous forme de macros, regroupés dans un fichier spécifique.

Une image est un tableau dont tous les éléments ont le même format de virgule fixe.

Les paramètres et précisions sont soit sous forme de macro lorsqu'ils sont statiques, soit en variable globale.

Les images à traiter sont dans des fichiers lus et écrits par l'intermédiaire de tableaux.

L'utilisation de la virgule fixe produit un 'bruit' de calcul et il est possible de mesurer l'écart à la référence algorithmique. Dans un premier temps, de façon à se focaliser sur la fonction, les paramètres de la virgule fixe sont assez large et permettent un fonctionnement de l'algorithme.

Remarque : Il est tentant d'avoir le même code pour la virgule flottante et la virgule fixe et de changer le type de données à l'aide de macros. A court terme cela est séduisant mais se révèle rapidement gênant car les codes sont très différents et les mélanger devient totalement contre-productif.

Code pour la HLS

Semblable au précédent, les mémoires étant identifiées et le maximum de paramètres sont statiques. Les interfaces au système sont identifiées :

— Registres de configuration
— Mémoire de données lues & écrites
— Mémoires pour les données intermédiaires

Pour le banc de test, les images d'entrée/sortie sont dans des fichiers luts/écrits depuis des tableaux.

1. Documentation : `/softslin/catapultc10_1b/Mgc_home/ac_datatypes_ref.pdf`

La vérification post-HLS est incorporée à la HLS et l'outil compare automatiquement les résultats produits par le code RTL généré et les résultats de l'exécution du code C/C++ pour la HLS.

Après HLS, la synthèse logique RTL permet de vérifier les performances (timing) ainsi que le coût matériel.

Emulation FPGA

Les paramètres sont placés soit dans des registres ou bien des signaux constants. Dans le premier cas, un logiciel viendra régler les valeurs des registres.

Toutes les étapes avant l'émulation peuvent être réalisées à partir d'images dites 'de référence', qui serviront à faire tous les tests. Ainsi, il est possible de travailler sans caméra le plus longtemps possible.

Les images sont placées dans des SRAM, soit par initialisation de la SRAM, soit à l'aide d'un logiciel qui vient placer les données en SRAM.

Remarque : L'émulation apporte peu d'informations supplémentaires par rapport à la validation post HLS, car le placement/routage n'introduit (quasiment) pas d'erreurs fonctionnelles. L'intérêt peut être de vérifier plus rapidement des situations qui sont longues à reproduire par simulation. Inversement, la détection d'un bug nécessite de reproduire la situation en simulation, ce qui peut s'avérer difficile vu le peu d'observabilité sur FPGA.

Validation sur FPGA et caméra

Les images proviennent d'une caméra.

— Les images d'entrée sont placées en SRAM directement par l'entrée caméra. La sortie est une SRAM dont le contenu est affiché. Pour différentes raisons, il est plus simple d'utiliser des SRAM double-port. Un port est pour l'entrée (ou la sortie), l'autre pour l'accélérateur. Du point de vue de l'accélérateur, la mémoire est simple port.

— Les images sont placées en DDR-SDRAM, par exemple à l'aide de DMA. Cette solution est la plus réaliste mais est difficile à mettre en oeuvre.

2.2 Images

2.2.1 Codage des images

Une image est une matrice de pixels, chaque pixel étant soit codé sur plusieurs composantes couleur (RGB), soit simplement par une luminance. Typiquement les composantes sont des entiers sur 8 bits, 16 bits, ou autre. En général les componsantes ne sont pas de valeurs signées.

Il existe des codages plus complexes (par exemple le codage 4.2.2) et les composantes peuvent être autre chose que RGB (YUV, etc...) mais ce n'est pas le propos de cette boîte à outils.

En général les images sont représentées en mémoire de deux façons :

— Les composantes sont regoupées par pixels, dans l'ordre (R0, G0, B0, R1, G1, B1, etc...)

— Les composantes sont séparées (tous les Rn, puis tous les Gn, les Bn, etc ...)

Réaliser un traitement d'image consiste généralement à calculer chacun des pixels d'une image de sortie en fonction des pixels d'une image d'entrée. De façon à simplifier la conception, nous ne traiterons que des images en luminance, sur une seule composante par pixel.

2.2.2 Stockage des images

En mémoire du calculateur, la matrice est stockée 'linéairement', c'est à dire que le pixel (i, j) se trouve à l'adresse relative

$$@(i, j) = i + j * t_x$$

Avec (t_x, t_y) les tailles horizontale et verticale de l'image.

Il existe d'autre schémas d'adressage que nous verrons au cours du projet.

Une image est stockées soit en SRAM soit en DDR-SDRAM. Les différences sont les suivantes :

— En DDR-SDRAM ; Pour accéder à un pixel il est nécessaire de passer par un bus système et par le contrôleur mémoire. Les DDR-SDRAM ont des débits élevés mais des latences élevées et ne sont efficaces que si l'on accède à plusieurs pixels dans des bursts. Il est nécessaire de mettre en place une mémoire de travail proche de l'accélérateur (mémoire cache, SPRAM, SRAM, etc...), dans laquelle on recopiera des zones de l'image.

— En SRAM ; Dans ce cas, la SRAM fourni un mot mémoire par cycle d'horloge, le cycle suivant la présentation de l'adresse. Les SRAM sont très rapides mais de quantité réduite.

Pour les projets, pour des raisons de simplicité, nous privilégierons les SRAM en sachant qu'une implémentation réaliste serait plutôt un stockage en DDR-SDRAM avec un mécanisme de cache plus ou moins spécialisé.

2.2.3 Format des images

Pour la conception d'accélérateurs par HLS, il est donc possible de considérer une image comme un tableau de taille $t_x * t_y$ et d'y accéder en calculant l'adresse de façon appropriée.

Pour la validation du code C/C++, l'image est chargée en mémoire avant le lancement de l'accélérateur. A cette fin, l'image pourra être lue depuis un fichier.

Les formats de fichier les plus simples sont :

— `RAW` ; Le fichier n'a pas d'en-tête, les valeurs binaires des pixels sont les unes après les autres et les composantes entrelacées.

L'entrelacement peut être :

  - Par pixel ; la séquence RGB répétée pour chaque pixel
    R0 G0 B0 R1 G1 B1
  - Par plan ; on trouve les composantes les unes après les autres
    R0 R1 B0 B1 G0 G1
- `PPM` ; Format simple à en-tête (voir `man ppm`), avec les données soit en texte ASCII soit en binaire, entrelacées par pixel. Les formats sont soit `PGM` (voir `man pgm`) pour les images en luminance (niveau de gris), soit `PPM` pour les images couleur.
  Exemple d'image PGM de taille $(2, 2)$:

```
P2
# Un commentaire
2 2
255
0 255 255 0
```

Ces types de fichier peuvent être produit de plusieurs façons. Par exemple :

- Logiciel `GIMP`, menu fichier->exporter->format `data`, `pgm` ou `ppm`
- Utilitaire `ImageMagick`, commande `convert`

  - `convert -monochrome -compress none image_entree.jpg image_sortie_ASCII.pgm`
  - `convert -monochrome image_entree.jpg image_sortie_BINAIRE.pgm`
  - `convert -monochrome image_entree.jpg GRAY:image_RAW_BINAIRE.raw`
  - `convert  image_entree.jpg RGB:image_RAW_BINAIRE_Couleur.raw`

Les utilitaires `ImageMagick` sont très efficaces et il est préférable de consulter leur documentation.

2.2.4 Accès aux images pour la validation HLS

Une fois l'image PGM ASCII formée, il est possible de la lire très simplement depuis n'importe quel langage, C ou Python. A l'inverse, l'écrire est aussi très facile. Les exemples suivant sont pour le format ASCII et peuvent être facilement transposés pour le format binaire.

En C

Exemple de code de lecture d'une image PGM ASCII (format `P2`) :

```
f=fopen("Mon_fichier.pgm", "r");
fscanf(f,"%s", format);
/* Lit la ligne qui contient la taille */
fgets(taille, 256, f);
/* mais passe les commentaires */
while (taille[0]=='#') fgets(taille, 256, f);
// lit la taille dans la chaine
sscanf(taille,"%d %d", &tx, &ty);
fscanf(f,"%s", temp);
```

```
for(int i=0;i<tx*ty*;i++)
  fscanf(f,"%d", &image[i]);
```

Dans cet exemple, on suppose que `image` est un pointeur sur un tableau d'entier préalablement alloué. Pour faire plus générique, on peut allouer l'image dynamiquement après avoir lut la taille. Il est possible de faire un peu plus élégant en C++.

En Python

Encore plus facile car il suffit de lire les lignes...

```
img=open('Mon_fichier.pgm');
format=img.readline()
line=img.readline()
while line[0] == '#' : line=img.readline()
(width,height) = [int(i) for i in img.readline().split()]
valmax=img.readline()
raster = []
for i in range(width*height):
    raster.append(int(img.read()))
img.close()
```

Il est possible de lire un fichier PGM binaire et le placer dans une variable de la façon suivante :

```
img=open('Mon_fichier.pgm');
format=img.readline()
line=img.readline()
while line[0] == '#' : line=img.readline()
(width,height) = [int(i) for i in img.readline().split()]
valmax=img.readline()
image_data=img.read(width*height)
img.close()
```

Pour lire un fichier RAW binaire :

```
img=open('Mon_fichier.pgm', 'b');
image_data_raw=img.read()
img.close()
```

L'écriture d'un fichier est le symétrique (voir documentation Python)

Ecriture/lecture en C++ d'une image en fichier

Pour lire une image ppm en C++ :

```
#define IMG_SIZE_0 320
#define IMG_SIZE_1 240
#define IMG_SIZE IMG_SIZE_0*IMG_SIZE_1

/* Lecture d'une image dans le tableau img_out */
 ac_fixed<DATA_WIDTH,DATA_WIDTH,false,AC_RND_INF,AC_SAT> img_in[IMG_SIZE];
...
 if (simg_in.is_open()) printf("file opened\n");
  simg_in.getline(type, 128);
  simg_in.getline(tmp, 128);
  while (tmp[0]== '#') simg_in.getline(tmp, 128);
  sscanf(tmp, "%d %d\n", &sx, &sy);
  printf( "Taille lue : %d %d\n", sx, sy);
  int level;
  simg_in>>level;
  int data;
  for(int i=0; i<sx*sy;i++)
   {
     simg_in >> data;
     img_in[i]= data;
   }
```

Pour écrire une image ppm en C++ :

```
/* Ecriture d'une image dans le tableau img_out */
ac_fixed<DATA_WIDTH,DATA_WIDTH,false,AC_RND_INF,AC_SAT> img_out[IMG_SIZE];
....
 simg_out << "P2" << endl;
 simg_out << IMG_SIZE_0<< " ";
 simg_out <<  IMG_SIZE_1<< endl;
 simg_out <<  255 << endl;
 for(int i=0; i<sx*sy;i++)
  {
   simg_out <<  img_out[i].to_int() << endl;
  }
```

2.2.5 Accès aux images dans le FPGA

En SRAM du FGPA

Bien entendu, pour accéder aux pixels de l'image en SRAM il faut que la SRAM contienne les pixels. L'objectif est de valider le traitement sur la même image que celle utilisée pour la simulation, avant d'utiliser des images en provenance de la caméra. Il existe plusieurs façons d'émuler une image qui proviendrait d'une caméra :

— La SRAM est initialisée par les valeurs des pixels (raw)
  — La SRAM est initialisée à la création de l'unité SRAM. Il faut créer un fichier au format `coe`, qui est la liste des valeurs séparées par une virgule. Le fichier `coe` pourra être produit par un script Python ou tout simplement à partir d'un fichier PGM. Le fichier au format `coe` sera utilisé à la création de la SRAM dans Vivado.
  — Une entité VHDL de la SRAM est générée à partir de l'image, avec les valeurs des pixels. Ici, la 'SRAM' est un signal dont la valeur initiale est la liste des pixels. Un tel fichier pourra être généré par un script Python.
  Voir section A.4
— La SRAM est initialisée par le logiciel du processeur
  Si l'on utilise une SRAM double port, l'un des ports peut être accédé par le processeur s'il est interfacé par une interface esclave sur le bus AXI. Le logiciel écrit la SRAM avec le contenu de l'image, ou bien la lit.

Génération d'un fichier coe

En python, générer un fichier au format **coe** est trivial.

```
print('memory_initialization_radix=10;')
print('memory_initialization_vector=')
print(',\n'.join(str(val) for val in image_data) + ';')
```

Vous pouvez aussi utiliser des base 2 ou 16, selon les besoins.

Logiciel 'bare-metal' et images

L'image peut être accédée depuis le logiciel (par exemple pour être copiée dans la SRAM de l'accélérateur) de plusieurs façon :

— Tableau initialisé
  Un tableau est initialisé par les valeurs des pixels de l'image à l'aide d'une variable globale.
  Le tableau déclaré est initialisé par les pixels. Un tel code peut être généré en Python à partir de l'image pgm/ppm.
— Objet initialisé
  De façon à soulager le compilateur, un fichier objet qui contient l'image au format RAW est incorporé à l'édition de lien en ajoutant une section (`.input_data` par ex.). Ensuite, l'image est accédée par une variable qui pointe vers le symbole associé.

  — Conversion du fichier image

```
$CROSS-objcopy -I binary -O elf32-littlearm -B arm \
               --rename-section .data=.input_data,alloc,load,readonly,data,contents
               image_RAW_BINARY.data image.o
```

en ayant affecté `$CROSS` par le préfixe du cross-compilateur

— Incorporation par linkerscript
Dans le linkerscript, ajouter une section avant la fin :
```
.input_data . : {
   . = ALIGN(64);
   _data_image_start = . ;
    image.o(.input_data)
   _data_image_end = . ;
} > memory_region
```
En prenant soin de remplacer les noms et de modifier `memory_region` selon l'entête du linker script (par exemple `ps7_ddr_0_S_AXI_BASEADDR` ou `ps7_ddr_0`)

— Accès depuis le C/C++
Déclarer une variable de même nom que le symbole dans la section ajoutée, puis utiliser un pointeur sur l'adresse de ce symbole.
```
extern unsigned int _data_image_start;
...
unsigned char *img=&_data_image_start;
...
p=img[x+tx*y]; // accès au pixel x,y  / tx = taille horizontale de l'image
```

— Fichier
L'image est lue dans un fichier accessible (par exemple sur carte SD). Il sera possible d'utiliser un code similaire à la lecture d'une image pgm, mais en tenant compte du fait qu'il n'y a pas d'allocation mémoire dynamique en bare-metal.

En général lorsque l'image est incorporée dans le logiciel elle sera automatiquement placée en DDR-SDRAM. Bien que cela soit rarement nécessaire, il est possible de forcer son placement dans une zone particulière à l'aide du *linker script*.

En logiciel sous Linux

Les techniques précédentes sont utilisées mais la lecture de fichiers est encore plus simple. Le code utilisé pour la simulation peut être réutilisé.

Accès aux images en DDR-SDRAM

Il est possible de placer les images en mémoire DDR-SDRAM externe au FPGA. Dans ce cas, il faut mettre en place les mécanismes d'accès aux données depuis l'accélérateur. L'accélérateur charge ses données depuis la DDR-SDRAM de deux façons :

— Il dispose de son propre DMA
Dans cette situation, l'accélérateur se charge de demander les zones de données au DMA.

— Le DMA est externe à l'accélérateur
Un mécanisme de synchronisation HW/SW permet à l'accélérateur de 'réclamer'

au logiciel les données à charger. Le logiciel organise le transfert des données. Cette méthode est relativement souple mais peu efficace.

Dans les deux cas, il faudra veiller à l'alignement des données en mémoire. Par exemple, le premier pixel d'une image (ou d'une ligne) doit être au début d'un mot mémoire car le DMA ne peut pas commencer par une donnée qui n'est pas alignée. Pour aligner les données en mémoire il faudra forcer les adresses des données sur des multiples de la taille du bus de donnée.

Il est possible de placer des données en DDR-SDRAM à l'initialisation du SW, à l'aide du linker-script, et de les aligner. Voir la section précédente sur le logiciel en bare-metal.

L'accès aux données en DDR-SDRAM en présence de Linux pose de sérieux problèmes car le DMA fonctionne en adresses physiques alors que les applications utilisent des adresses virtuelles et les données des tableaux peuvent ne pas avoir des adresses physiques continues. Il faut réussir à gérer des données de plages mémoire physique contigües et cela nécessite l'utilisation de fonctions noyaux spécifiques, utilisable seulement par des *module* ou *driver*. Vu la durée des projets, cette technique est exclue.

2.3 Gestion de la virgule fixe

La documentation de la virgule fixe *CatapultC* est :
`/softslin/catapultc10_5c/Mgc_home/shared/pdfdocs/ac_datatypes_ref.pdf`

Les fichiers de la bibliothèque sont :
`/softslin/catapultc10_5c/Mgc_home//shared/include//shared/include/ac_fixed.h`
`/softslin/catapultc10_5c/Mgc_home//shared/include//shared/include/ac_int.h`

Attention : Vous pouvez copier ces fichiers et les utiliser pour compiler vos programmes avec *gcc/g++*, **mais** vous devez les enlever du répertoire projet lorsque vous passerez à *CatapultC* car ces copies entreront en conflit avec les fichiers originaux. Une solution est de les mettre dans un autre répertoire et faire un `Makefile` qui va bien.

Pour résumer le document "Algorithmic C Datatypes" de MentorGraphics, le format des nombres en virgule fixe est

```
ac_fixed<W,I,sign>
```

avec `W` la taille totale et `I` le nombre de bits de la partie entière. `sign` indique si le nombre est signé `true` ou non-signé `false`. Le nombre de bits à droite est donc `W-I`. Si le nombre est signé, le bit de signe est compté dans les `I` bits à gauche de la virgule.

Par exemple, pour déclarer un nombre en virgule fixe, le template précédent est utilisé pour le type de la variable :

```
ac_fixed<16,3,true> v;
```

Déclare une variable `v` de taille totale 16 bits, dont 3 à gauche de la virgule (bit de signe inclu) et 13 à droite de la virgule. La valeur sera considérée comme un nombre en complément à deux.

Cette bibliothèque gère automatiquement les changements de type et conversion depuis/vers les nombres en virgule flottante. Les opérateurs arithmétiques sont 'surchargés' et gèrent automatiquement les décalages nécessaires pour réaliser des additions ou multiplication sur des nombres de formats différents. Il est également possible de réaliser des opérations de troncature/arrondi automatiquement, par déclaration des bons formats et affectations entre variables de formats différents.

Par exemple pour réaliser une troncature de `a` vers `b` :

```
ac_fixed<a_W, a_I, a_sign> a;
ac_fixed<b_W, b_I, b_sign> b;
b=a;
```

Par défaut, `b` est affecté par troncature (ou extension de zéro si `a` a moins de bits à droite de la virgule). Il est possible de spécifier le mode d'affectation (arrondi/troncature) avec deux paramètres supplémentaires du template (voir documentation).

Les opérations arithmétique de base suivent le même principe : les arguments sont automatiquement alignés et le résultat mis dans le format de la variable destination.

```
ac_fixed<a_W, a_I, a_sign> a;
ac_fixed<b_W, b_I, b_sign> b;
ac_fixed<c_W, c_I, c_sign> c;
b=a;
c=a+b;
c=m*b;
```

De façon à gérer simplement la virgule fixe, l'expérience montre qu'il est préférable de gérer le paramétrage dans un fichier de configuration :

```
/* Fichier  imgproc_vfix_config.h /*
#define IMGPROC_A_PS 20, 10, true, AC_RND

/* Fichier  imgproc.c */
ac_fixed<IMGPROC_A_PS> a;
```

2.3.1 Initialisation des constantes en virgule fixe

Il est possible d'initialiser des constantes directement par leur valeur en virgule flottante et le template fait automatiquement un transtypage. Par exemple :

```
ac_fixed<16,3,true> v=5.324567;
ac_fixed<16,3,true> w=1.0/3.0;
```

Il est aussi possible de les affecter dans le code :

```
ac_fixed<16,3,true> v;
v=5.324567;
```

Il est aussi possible d'initialiser des tableaux, dont tous les éléments ont le même format de virgule fixe :

```
ac_fixed<16,3,true> t[]={5.324567, -.459046, 0.55111, 6.222, -1.9785};
```

Les affectation de valeur en virgule flottante par un variable en firgule fixe se fait par simple affectation. On peut aussi placer des champs en virgules fixes dans des structures, etc ... .

Remarque : Pour initialiser une mémoire par un tableau de constantes, il faut bien spécifier à Catapult-C que la variable est allouée à une RAM ou à une ROM. Cela peut nécessiter de valider l'utilisation de RAM ou ROM lors de l'étape de paramétrisation de la technologie.

2.3.2 Gestion des images

Une image est un tableau et le format de la virgule fixe s'applique à tous les éléments du tableau :

```
/* Fichier  imgproc_vfix_config.h /*
#define IMGPROC_IMAGE_IN_SIZE_0 640
#define IMGPROC_IMAGE_IN_SIZE_1 480
#define IMGPROC_IMAGE_IN_P 20, 10, true, AC_RND

#define IMGPROC_IMAGE_IN_SIZE IMGPROC_IMAGE_IN_SIZE_0*IMGPROC_IMAGE_IN_SIZE_1


/* Fichier  imgproc.c /*
ac_fixed<IMGPROC_IMAGE_IN_P> image_in[IMAGE_IN_SIZE];
```

La HLS produira automatiquement une interface à une mémoire de taille `IMGPROC_IMAGE_IN_SIZE`, avec des mots mémoire de taile `IMGPROC_IMAGE_IN_P(W)`. Selon les besoins, cette mémoire pourra être simple ou double port.

Remarque : L'utilisation de préfixe dans les paramètres permet d'éviter les conflits entre modules.

Remarque : L'utilisation du numéro d'axe (0, 1, etc ...) au lieu de leurs noms usuels (X, Y, etc ...), permet d'automatiser le nommage et de scripter la génération de paramètres

2.3.3 Initialisation des images

Le tableau stockant une image peut être initialisé par la liste des valeurs des pixels. La constante d'initialisation peut être facilement générée par un script Python.

Remarque : Pour initialiser une mémoire par un tableau de constantes, il faut bien spécifier à Catapult-C que la variable est allouée à une RAM ou à une ROM. Cela peut nécessiter de valider l'utilisation de RAM ou ROM lors de l'étape de paramétrisation de la technologie.

2.4 Paramétrage des IP

2.4.1 Motivation

Il est assez commun que le format d'une variable intermédiaire dépende de celui d'une autre variable. Par exemple, si on calcule $c = a * b + e$ et que l'on souhaite garder toute la précision de $a * b$ avant de faire la troncature pour produire c, on peut avoir :

```
/* Fichier  imgproc_vfix_config.h /*
#define IMGPROC_A_PS 20, 10, true
#define IMGPROC_B_PS 20, 10, true
#define IMGPROC_AB_PS 40, 10, true
#define IMGPROC_E_PS 10, 5, true

#define IMGPROC_C_PS 20, 10, true, AC_RND
...
/* Fichier  imgproc.c /*
ac_fixed<IMGPROC_A_PS> a;
ac_fixed<IMGPROC_B_PS> b;
ac_fixed<IMGPROC_E_PS> e;
...
ac_fixed<IMGPROC_AB_PS> ab;
ab=a*b;
c=ab+e;
```

Dans ce cas, la précision de `ab` dépend de celle de `a` et de `b`.

Il peut être utile d'écrire un script Python qui génère automatiquement la configuration de la virgule fixe en calculant automatiquement tout ce qu'il est possible de calculer, puis génèrer un fichier de configuration de la virgule fixe.

Chapitre 3

Plateformes de références

3.1 Plateforme Zybo Z7 Processing & Affichage

3.1.1 Configuration de la session

Les fichiers de configuration à sourcer :
— `> source bash_mentor_21`
— `> source bash_vivado_20`

3.1.2 Projet Vivado

Cette section décrit l'utilisation du projet de référence `Z7_HDMI_proc` pour *Vivado Vitis* (**??**). . Le projet `Z7_HDMI_proc` permet d'utiliser la sortie vidéo HDMI avec un accélérateur de traitement vidéo matériel.

Ce projet, illustré figure 3.1, permet de connecter :
— Une unité de traitement vidéo (accélérateur), qui peut etre conçue à l'aide de *CatapultC*
— Une mémoire de sortie vidéo
— Une sortie vidéo HDMI

Le projet constitué du processeur et du module `HDMI_proc` est réalisé en schématique *Vivado Vitis*, afin de générer la configuration du FPGA et programmer le processeur ARM. L'accélérateur est connecté à l'interface vidéo dans un module dénommé `HDMI_proc`, dans le répertoire `ip_repo`.

3.1.3 CatapultC

Pour configurer *CatapultC*, voir A.2.

L'accélérateur de démonstration est dans `Catapult/CC_demo/DispProcTest`. `DispProcTest.cpp` génère une image en niveau de gris allant de 0 sur le coin supérieur en augmentant selon la diagonale. Son interface correspond à celle des mémoires *Vivado*, sur 8 bit.

Les fichiers utiles sont :
— Fichier source : `Catapult/CC_demo/DispProcTest/DispProcTest.cpp`

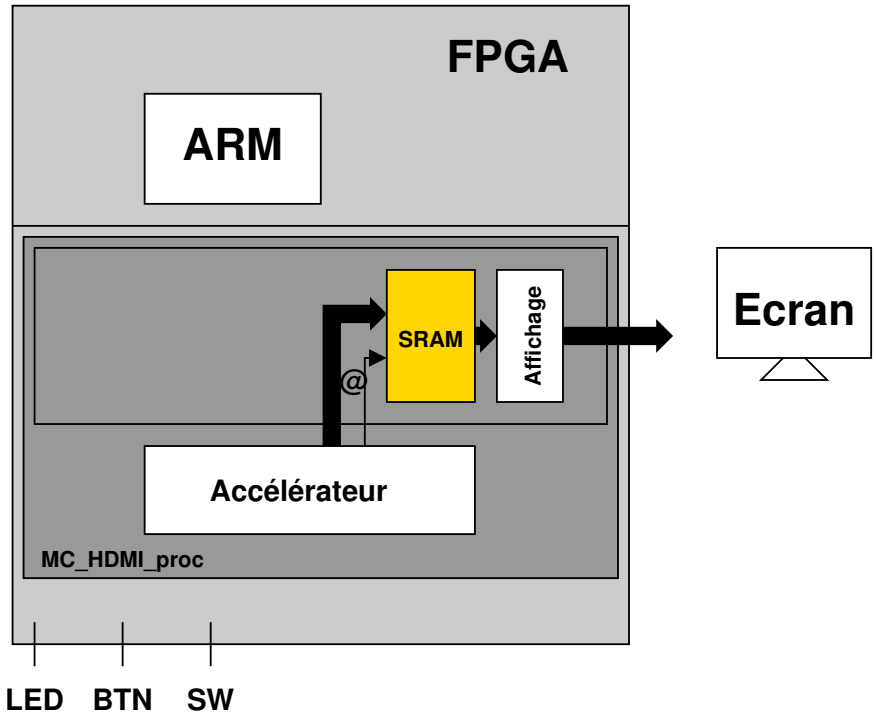


FIGURE 3.1 – Projet `Z7_HDMI_proc`

— Fichier de configuration et script de synthèse HLS : `Catapult/CC_demo/DispProcTest/directiv`

Pour lancer le projet CatapultC ;

— Créer un projet CatapultC *'File-> New Project -> valider'*
— Dans *'Tools - Set Options - Flow - Precision RTL '*, si ce n'est déjà fait, décochez l'onglet *'Add IO Pads'* (☞ très important !!), puis *'Apply & Save '*.
— Lancer le script de configuration et synthèse : *'File -> run script -> choisir directives.tcl'*
— Générer la netlist : dans l'onglet *'Project File -> Synthesis -> Synthesize RTL -> (bouton droit) Launch Precision Batch'*

Ceci produit une netlist au format `edf`, que vous trouverez à l'aide de la commande :

```
> find .  -name '*edf'
```

Comme *CatapultC* génère un nouveau répertoire à chaque modification, vérifiez bien que vous avez la bonne version.

Une fois le `edf` généré, il vous reste à l'intégrer au projet *Vivado*

3.1.4 Gestion de l'IP

L'unité de calcul est mise dans un format qui permet son intégration à *Vivado Vitis*. Les fichiers de *définition de l'IP* pour *Vivado Vitis* sont dans `Zynq/Z7/Z7_Demo_Cam/ip_repo/HDMI_proc`. **Ce ne sont pas les fichiers pour le projet**

FGPA, seulement la gestion du module.

Le fichier le plus important est `src/HDMI_proc.vhd`. Ce code VHDL permet de connecter le module de traitement `DispProcTest.edf` au système d'acquisition vidéo et affichage `HDMI_RAM_syn.edf`. Ce dernier permet d'afficher des images 320x240 en luminance, sur 8 bits. Me demander pour une autre résolution, en sachant qu'il n'y a que 4 Mbit de RAM sur le FPGA.

Pour mettre à jour l'unité de calcul, il y a plusieurs posssibilités :

- L'interface est exactement identique à `DispProcTest` :
  - soit remplacer la netlist `DispProcTest.edf` par la nouvelle, en gardant le nom `DispProcTest.edf`
  - soit la renommer et modifier le nom dans le code VHDL. Dans ce cas, il faut aussi modifier `component.xml` et remplacer `DispProcTest` par le nouveau nom, dans tous les fichiers, afin que *Vivado* retrouve vos netlists et codes VHDL.
- L'interface est différente et vous voulez ajouter des entrées/sortie, ou interface aux LED et boutons
  - Modifier le VHDL de `HDMI_proc.vhd`
  - Si besoin ajouter des fichiers supplémentaires dans `component.xml`

Si vous avez besoin de mémoires avec des constantes, le plus simple est de déclarer des tableaux initialisés par les bonnes valeurs dans le code CPP pour la HLS, Catapult génèrera automatiquement les mémoire initialisées avec les bonnes valeurs.

Vous pouvez aussi intégrer des IP *Vivado Vitis* dans votre projet (SRAM etc). Si besoin, me demander au cas par cas.

3.1.5 Gestion du projet sous *Vivado*

Le projet de la plateforme pour programmer le FPGA est `Zynq/Z7/Z7_Demo_Cam/Z7_HDMI_proc_20`.

Pour utiliser le projet *Vivado Vitis* et programmer le FPGA :

- Avec *Vivado*, ouvrir le fichier `Zynq/Z7/Z7_Demo_Cam/Z7_HDMI_proc_20/Z7_HDMI_proc.xpr`
- Cliquer *'Project Manager'* dans l'onglet de gauche
- Ensuite, dans la barre de fenêtre du dessus, *'Tools -> Report-> Report IP Status'*
- Dans l'onglet *'IP Status'* qui apparait en bas, cliquer *'Upgrade Selected'*, répondre Yes à toute les questions, et *'Generate'* output products.
- Lorsqu'il a terminé,
- *'Program and debug -> generate Bitstream'*
- Brancher la carte Z7 sur le port USB et la sortie HDMI
- *'Program and debug -> Open Hardware Manager -> Open Target'*
- *'Program and debug -> Open Hardware Manager -> Program Device'* celui qui est proposé, (au passage, vérifiez que le fichier du bitstream est le bon)

Bravo, il vous reste à debugger!!

Note : le projet `Z7_HDMI_proc` nécessite d'appuyer sur le bouton à droite de la Zybo, connecté au `reset` de l'accélérateur.

3.1.6 Mise à jour

Vous pouvez faire toutes les étapes précédentes sans quitter les logiciels. Lorsque vous modifiez la netlist dans le répertoire de l'IP, *Vivado Vitis* vous propose automatiquement une mise à jour du projet vous pouvez reprendre toute la procédure de génération du bitstream. Lorsque vous effectuez la mise à jour du fichier EDF, l'IP devrait apparaître cochée dans la fenêtre 'IP Status'. Si ce n'est pas le cas, recommencez *'Tools -> Report-> Report IP Status'*.

Note : si la mise à jour ne se fait toujours pas, allez dans le répertoire des fichiers de l'IP, sours répertoire `src`, et tapez la commande bash `touch *`

3.2 Plateforme Zybo Z7 Caméra, Processing & Affichage

3.2.1 Configuration de la session

Les fichiers de configuration à sourcer :
— `> source bash_mentor_21`
— `> source bash_vivado_20`

3.2.2 Projet Vivado

Cette section décrit l'utilisation du projet de référence `Z7_MC_HDMI_proc` pour *Vivado* (A.1). . Le projet `Z7_MC_HDMI_proc` permet d'utiliser une caméra et la sortie vidéo HDMI avec un accélérateur de traitement vidéo matériel.

Ce projet, illustré figure 3.2, permet de connecter :
— Une caméra avec protocole MIPI-CSI
— Une mémoire vidéo d'entrée
— Une unité de traitement vidéo (accélérateur), de mémoire à mémoire, qui peut etre conçue à l'aide de *CatapultC*
— Une mémoire de sortie vidéo
— Une sortie vidéo HDMI

Ce projet nécessite le processeur ARM car la caméra est configuré par un logiciel 'bare-metal' sur le processeur ARM du Zynq, par le protocole I2C.

Le projet constitué du processeur et du module `MC_HDMI_proc` est réalisé en schématique *Vivado*, afin de générer la configuration du FPGA et programmer le processeur ARM. L'accélérateur est connecté aux interfaces caméra et vidéo dans un module dénommé `MC_HDMI_proc`, dans le répertoire `ip_repo`.

3.2.3 CatapultC

Pour configurer *CatapultC*, voir A.2.

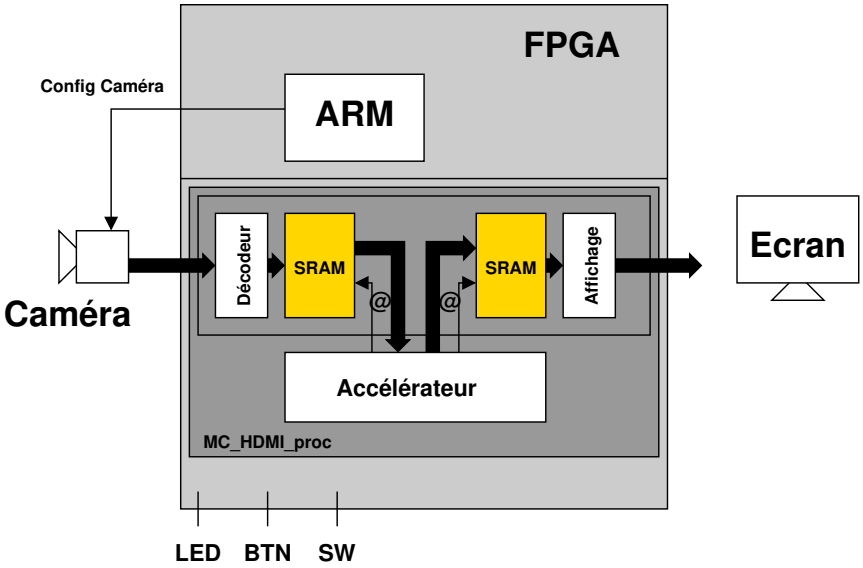


FIGURE 3.2 – Projet `Z7_MC_HDMI_proc`

L'accélérateur de démonstration est dans `Catapult/CC_demo/ImgProcTest`. Le code du fichier `ImgProcTest.cpp` 'inverse' la luminosité des pixels au dessus de la bissectrice. Ce projet écrit les pixels de sortie en lisant la mémoire vidéo d'entrée. Son interface correspond à celle des mémoires *Vivado*, sur 8 bit.

Les fichiers utiles sont :

— Fichier source : `Catapult/CC_demo/ImgProcTest/ImgProcTest.cpp`

— Fichier de configuration et script de synthèse HLS : `Catapult/CC_demo/ImgProcTest/directives.tcl`

Pour lancer le projet CatapultC ;

— Créer un projet CatapultC *'File-> New Project -> valider'*

— Dans *'Tools - Set Options - Flow - Precision RTL '*, si ce n'est déjà fait, décochez l'onglet *'Add IO Pads'* (☞ très important !!), puis *'Apply & Save '*.

— Lancer le script de configuration et synthèse : *'File -> run script -> choisir directives.tcl'*

— Générer la netlist : dans l'onglet *'Project File -> Synthesis -> Synthesize RTL -> (bouton droit) Launch Precision Batch'*

Ceci produit une netlist au format `edf`, que vous trouverez à l'aide de la commande :

```
> find .  -name '*edf'
```

Comme *CatapultC* génère un nouveau répertoire à chaque modification, vérifiez bien que vous avez la bonne version.

Une fois le `edf` généré, il vous reste à l'intégrer au projet *Vivado*

3.2.4 Mise à jour de l'accélérateur

L'unité de calcul est mise dans un format qui permet son intégration à *Vivado*. Les fichiers de *définition de l'IP* pour *Vivado* sont dans MC_HDMI_proc. **Ce ne sont pas les**

fichiers pour le projet FGPA, seulement la gestion du module.

Le fichier le plus important est `src/MC_HDMI_proc.vhd`. Ce code VHDL permet de connecter le module de traitement `ImgProcTest.edf` au système d'acquisition vidéo et affichage `MC_HDMI_RAM_syn.edf`. Ce dernier permet d'acquérir des images 320x240 en luminance, sur 8 bits. Me demander pour une autre résolution, en sachant qu'il n'y a que 4 Mbit de RAM sur le FPGA.

Pour mettre à jour l'unité de calcul, il y a plusieurs posssibilités :

- L'interface est exactement identique à `ImgProcTest` :
  - soit remplacer la netlist `ImgProcTest.edf` par la nouvelle, en gardant le nom `ImgProcTest.edf`
  - soit la renommer et modifier le nom dans le code VHDL. Dans ce cas, il faut aussi modifier `component.xml` et remplacer `ImgProcTest` par le nouveau nom, dans tous les fichiers, afin que *Vivado* retrouve vos netlists et codes VHDL.
- L'interface est différente et vous voulez ajouter des entrées/sortie, ou interface aux LED et boutons
  - Modifier le VHDL de `MC_HDMI_proc.vhd`
  - Si besoin ajouter des fichiers supplémentaires dans `component.xml`

Si vous avez besoin de mémoires avec des constantes, le plus simple est de déclarer des tableaux initialisés par les bonnes valeurs dans le code CPP pour la HLS, Catapult génèrera automatiquement les mémoire initialisées avec les bonnes valeurs.

Vous pouvez aussi intégrer des IP *Vivado* dans votre projet (SRAM etc). Si besoin, me demander au cas par cas.

Note : `MC_HDMI_RAM_syn.edf` fournit l'horloge de l'unité de calcul et également un signal `bypass` qui permet d'afficher directement l'image d'entrée sur la sortie vidéo.

3.2.5 Gestion du projet sous *Vivado Vitis*

Le projet de la plateforme pour programmer le FPGA est `Zynq/Z7/Z7_Demo_Cam/Z7_MC_HDMI_proc_20`.

Pour utiliser le projet *Vivado Vitis* et programmer le FPGA :

- Avec *Vivado*, ouvrir le fichier `Zynq/Z7/Z7_Demo_Cam/Z7_MC_HDMI_proc_20/Z7_MC_HDMI_proc.xpr`
- Cliquer *'Project Manager'* dans l'onglet de gauche
- Ensuite, dans la barre de fenêtre du dessus, *'Tools -> Report-> Report IP Status'*
- Dans l'onglet *'IP Status'* qui apparait en bas, cliquer *'Upgrade Selected'*, répondre Yes à toute les questions, et *'Generate'* output products.
- Lorsqu'il a terminé,
- *'Program and debug -> generate Bitstream'*
- Brancher la carte Z7 sur le port USB et la sortie HDMI
- *'Program and debug -> Open Hardware Manager -> Open Target'*
- *'Program and debug -> Open Hardware Manager -> Program Device'* celui qui est proposé, (au passage, vérifiez que le fichier du bitstream est le bon)

Afin de lancer le logiciel de configuration de la caméra sur le processeur ARM, il est nécessaire de passer par l'environnement de programmation du logiciel. Dans *Vivado Vitis* :

- Exporter la configuration du matériel vers l'IDE de programmation
  'File - Export - Export Hardware- Yes', sélectionner *Include Bitstream' -> Next*, Vérifier le chemin du fichier exporté, pour qu'il soit dans le chemin du projet actuel
- Lancer l'IDE *'Tools - Launch VITIS IDE'*
  Vitis IDE vous demande de sélectionnenr un workspace. Normalement il y a un workspace
  `Zynq/Z7/Z7_Demo_Cam/Z7_MC_HDMI_proc_20/Z7_MC_HDMI_proc_wk`.
- Gestion du projet logiciel
  Le projet logiciel est décomposé en deux :
  - une gestion du logiciel de base associé à la plateforme matérielle, automatiquement mis à jour, identifié par une icône verte
  - votre projet, identifié par une icône bleue
- Mise à jour de la plateforme matérielle
  - Sur l'icône verte, *'Bouton Droit -> Update Hardware Specification'*, vérifiez que le chemin correspond au fichier `.xsa` exporté dans *Vivado Vitis*.
  - Sur l'icône verte, *'Bouton Droit -> Build Project'*
- Mise à jour du logiciel
  Après une mise à jour de la plateforme ou une édition du logiciel
  - Sur l'icône bleue, *'Bouton Droit -> Build Project'*
- Exécution du logiciel
  - Sur l'icône bleue, *'Bouton Droit ->'Run As -> Launch on Hardware'*.

Bravo, il vous reste à debugger !!

Il est possible d'interagir avec le logiciel par le port USB/Série, voir A.3

3.2.6 Mise à jour

Vous pouvez faire toutes les étapes précédentes sans quitter les logiciels. Lorsque vous modifiez la netlist dans le répertoire de l'IP, *Vivado Vitis* vous propose automatiquement une mise à jour du projet vous pouvez reprendre toute la procédure de génération du bitstream. Lorsque vous effectuez la mise à jour du fichier EDF, l'IP devrait apparaître cochée dans la fenêtre 'IP Status'. Si ce n'est pas le cas, recommencez *'Tools -> Report-> Report IP Status'*.

Note : si la mise à jour ne se fait toujours pas, allez dans le répertoire des fichiers de l'IP, sours répertoire `src`, et tapez la commande bash `touch *`

3.2.7 SDK & bugs

Si SDK inidique une erreur au moment de lancer le logiciel, vous pouvez éteindre le FPGA, charger le bitstream et relancer le logiciel.

3.3 Plateforme Zybo Z7 Processing & Affichage, sur bus AXI

3.3.1 Configuration de la session

Les fichiers de configuration à sourcer :

— `> source bash_mentor_21`

— `> source bash_vivado_20`

3.3.2 Projet Vivado

Cette section décrit l'utilisation du projet de référence `Z7_ProcHDMI_axi_20` pour *Vivado Vitis* (**??**). . Le projet `Z7_ProcHDMI_axi_20` permet d'utiliser la sortie vidéo

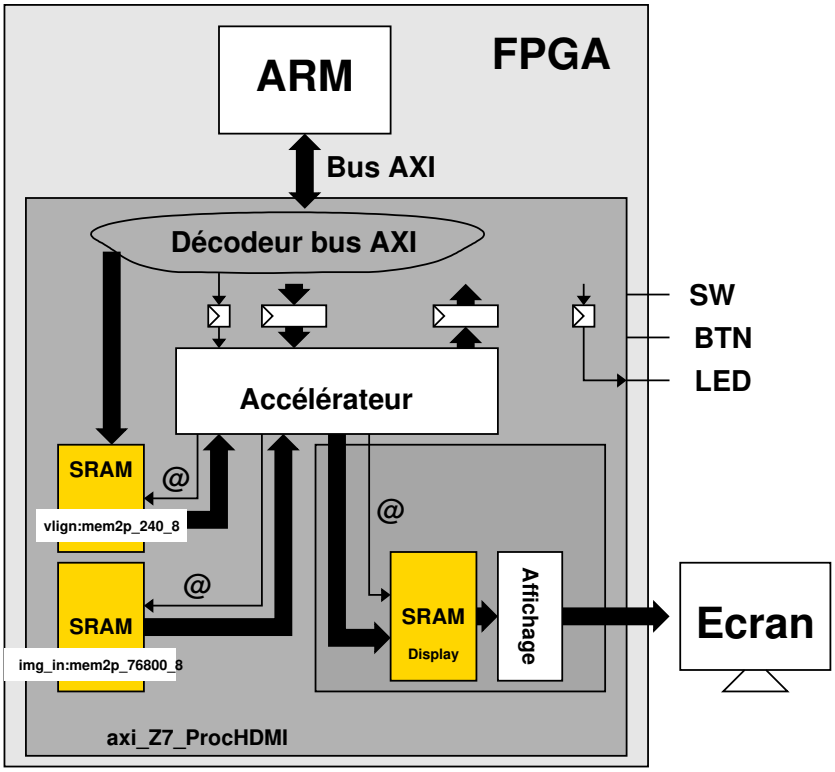


FIGURE 3.3 – Projet `Z7_ProcHDMI_axi_20`

HDMI avec un accélérateur de traitement vidéo matériel.

Ce projet, illustré figure 3.3, permet de connecter :

— Une unité de traitement vidéo (accélérateur), qui peut etre conçue à l'aide de *CatapultC*

— L'unité de traitement prend ses données depuis des mémoires et registres contrôlés par le logiciel embarqué sur le processeur ARM

— Une mémoire de sortie vidéo

— Une sortie vidéo HDMI

Le projet constitué du processeur et du module `axi_Z7_ProcHDMI`est réalisé en schématique *Vivado Vitis*, afin de générer la configuration du FPGA et programmer le processeur ARM. L'accélérateur est connecté à l'interface vidéo dans un module dénommé `axi_Z7_ProcHDMI`, dans le répertoire `ip_repo`.

3.3.3 CatapultC

Pour configurer *CatapultC*, voir A.2.

L'accélérateur de démonstration est dans `Catapult/CC_demo/ImgProcParamTest`. Le code du fichier `ImgProcParamTest.cpp` écrit les pixels de sortie `img_out` en lisant la mémoire vidéo d'entrée `img_in`.

— Selon le registre de contrôle `ctl`, 'inverse' la luminosité des pixels au dessus de la bissectrice

— Sur la colonne du milieu, remplace les pixels par le contenu de la mémoire `vlign_in`

— Calcule la moyenne des pixels et retourne la valeur par le registre `avg`

Les interfaces aux mémoires correspondent aux mémoires *Vivado*, sur 8 bit.

Les fichiers utiles sont :

— Fichier source : `Catapult/CC_demo/ImgProcParamTest/ImgProcParamTest.cpp`

— Fichier de configuration et script de synthèse HLS : `Catapult/CC_demo/ImgProcParamTest/directives`

Pour lancer le projet CatapultC ;

— Créer un projet CatapultC *'File-> New Project -> valider'*

— Dans *'Tools - Set Options - Flow - Precision RTL '*, si ce n'est déjà fait, décochez l'onglet *'Add IO Pads'* (☞ très important !!), puis *'Apply & Save '*.

— Lancer le script de configuration et synthèse : *'File -> run script -> choisir directives.tcl'*

— Générer la netlist : dans l'onglet *'Project File -> Synthesis -> Synthesize RTL -> (bouton droit) Launch Precision Batch'*

Ceci produit une netlist au format `edf`, que vous trouverez à l'aide de la commande :

```
> find .  -name '*edf'
```

Comme *CatapultC* génère un nouveau répertoire à chaque modification, vérifiez bien que vous avez la bonne version.

Une fois le `edf` généré, il vous reste à l'intégrer au projet *Vivado*

3.3.4 Gestion de l'IP

L'unité de calcul est mise dans un format qui permet son intégration à *Vivado Vitis*. Les fichiers de *définition de l'IP* pour *Vivado Vitis* sont dans `Zynq/Z7/Z7_Demo_Cam/ip_repo/axi_Z7_ProcHDMI`. **Ce ne sont pas les fichiers pour le projet FGPA, seulement la gestion du module.**

Le fichier le plus important est `src/io_video/axi_Z7_ProcHDMI_S00_AXI.vhd`. Ce code VHDL permet de connecter le module de traitement `ImgProcParamTest.edf`

au bus AXI et à l'affichage `HDMI_RAM_syn.edf`. Ce dernier permet d'afficher des images 320x240 en luminance, sur 8 bits.

L'interface au bus AXI est constitué d'un décodeur d'adresse locale qui permet d'identifier la destination d'une écriture ou lecture :

- mémoire `img_in`
- mémoire `vlign_in`
- registres `avg`, `ctl`
- d'effectuer un reset de l'IP, de gérer les LED et boutons poussoirs

Pour mettre à jour l'unité de calcul, il y a plusieurs posssibilités :

- L'interface est exactement identique à `DispProcParamTest` :
  - soit remplacer la netlist `DispProcParamTest.edf` par la nouvelle, en gardant le nom `DispProcParamTest.edf`
  - soit la renommer et modifier le nom dans le code VHDL. Dans ce cas, il faut aussi modifier `component.xml` et remplacer `DispProcParamTest` par le nouveau nom, dans tous les fichiers, afin que *Vivado* retrouve vos netlists et codes VHDL.
- L'interface est différente et vous voulez ajouter des entrées/sortie, ou interface aux LED et boutons
  - Modifier le VHDL de `src/io_video/axi_Z7_ProcHDMI_S00_AXI.vhd`
  - Si besoin ajouter des fichiers supplémentaires dans `component.xml`

Si vous avez besoin de mémoires avec des constantes, le plus simple est de déclarer des tableaux initialisés par les bonnes valeurs dans le code CPP pour la HLS, Catapult génèrera automatiquement les mémoire initialisées avec les bonnes valeurs.

Vous pouvez aussi intégrer des IP *Vivado Vitis* dans votre projet (SRAM etc). Si besoin, me demander au cas par cas.

3.3.5 Gestion du projet sous *Vivado*

Le projet de la plateforme pour programmer le FPGA est `Zynq/Z7/Z7_Demo_Cam/Z7_ProcHDMI_20`.

Pour utiliser le projet *Vivado Vitis* et programmer le FPGA :

- Avec *Vivado*, ouvrir le fichier `Zynq/Z7/Z7_Demo_Cam/Z7_ProcHDMI_20/Z7_ProcHDMI_axi_20.xpr`
- Cliquer *'Project Manager'* dans l'onglet de gauche
- Ensuite, dans la barre de fenêtre du dessus, *'Tools -> Report-> Report IP Status'*
- Dans l'onglet *'IP Status'* qui apparait en bas, cliquer *'Upgrade Selected'*, répondre Yes à toute les questions, et *'Generate'* output products.
- Lorsqu'il a terminé,
- *'Program and debug -> generate Bitstream'*
- Brancher la carte Z7 sur le port USB et la sortie HDMI
- *'Program and debug -> Open Hardware Manager -> Open Target'*
- *'Program and debug -> Open Hardware Manager -> Program Device'* celui qui est proposé, (au passage, vérifiez que le fichier du bitstream est le bon)

Afin de lancer le logiciel de configuration de la caméra sur le processeur ARM, il est nécessaire de passer par l'environnement de programmation du logiciel. Dans *Vivado Vitis* :

- Exporter la configuration du matériel vers l'IDE de programmation
  'File - Export - Export Hardware- Yes', sélectionner *Include Bitstream' -> Next*, Vérifier le chemin du fichier exporté, pour qu'il soit dans le chemin du projet actuel
- Lancer l'IDE *'Tools - Launch VITIS IDE'*
  Vitis IDE vous demande de sélectionnenr un workspace. Normalement il y a un workspace
  `Zynq/Z7/Z7_Demo_Cam/Z7_ProcHDMI_20/Z7_ProcHDMI_axi_20_wk`.
- Gestion du projet logiciel
  Le projet logiciel est décomposé en deux :
  - une gestion du logiciel de base associé à la plateforme matérielle, automatiquement mis à jour, identifié par une icône verte
  - votre projet, identifié par une icône bleue
- Mise à jour de la plateforme matérielle
  - Sur l'icône verte, *'Bouton Droit -> Update Hardware Specification'*, vérifiez que le chemin correspond au fichier `.xsa` exporté dans *Vivado Vitis*.
  - Sur l'icône verte, *'Bouton Droit -> Build Project'*
- Mise à jour du logiciel
  Après une mise à jour de la plateforme ou une édition du logiciel
  - Sur l'icône bleue, *'Bouton Droit -> Build Project'*
- Exécution du logiciel
  - Sur l'icône bleue, *'Bouton Droit ->'Run As -> Launch on Hardware'*.

Bravo, il vous reste à debugger !!
Il est possible d'interagir avec le logiciel par le port USB/Série, voir A.3

Annexe A

Annexes

A.1 Vivado

Pour pouvoir utiliser Vivado :
`> source /softslin/vivado_17.1/Vivado/2017.1/settings64.sh`

A.2 CatapultC

Pour faire son projet CatapultC copier `/tp-fmr/smancini/SLE/Projets/bash_mentor` ou `/tp-fmr/smancini/SEI_SoC_CNN/bash_mentor` puis `> source bash_mentor`. Catapult se lance par `> catapult`

A.2.1 Commentaires sur CC

A ce projet de base, vous pouvez
— Si besoin, ajouter des signaux de controle start/done/ready depuis CatapultC *'Mapping -> Solution cocher start/done/ready'*.
Ces signaux vous servirons à lancer l'unité lorsque vous en avez besoin. Il est possible de les connecter aux boutons poussoirs et switch de la carte
— Ajouter vos propres interface
  — Données et paramètres (en provenance de mémoires ou du logiciel)
  — Autres tableaux
  — controle (switch et boutons) ou debug (affichage sur LED)

A.3 Zybo

A.3.1 Port USB/Série

Le processeur ARM de la Zybo peut être connecté à un PC par USB et il est possible d'interagir par une console texte sur le port USB/Série.
La commande :

`> minicom -D /dev/ttyUSB0 (ou /dev/ttyUSB1)` Ceci permet l'affichage des `printf` du code sur le processeur ARM. Il est possible d'interagir avec le code ARM en lisant le port USB/Série par des `scanf` dans le code du processeur ARM. Dans de cas `minicom` transmet sur le port USB/Série le texte saisi par l'utilisateur et le code ARM le "récupère" ensuite.

Configurer minicom pour un débit de 115200b/s et sans contrôle de flux.

A.4 Génération de RAM

Exemple de code d'une SRAM générée avec un contenu initialisé. On ajoutera les bibliothèques adéquates et les termes entre $ seront remplacés par les noms ou listes de valeurs. Si besoin le type de `addr` sera adapté.

```
entity ${ram_name} is
  generic(
    --parameters size of the memory and width of the words
    --see in the manual for all possobilities
    -- total size of the memory is cellCount*wordSize
    cellCount : integer := ${ram_cc};  -- number of ram entries
    wordSize : integer := ${ram_ws};   -- size of ram data word
  );
  port( clk : in std_logic;
        addr : integer;
        din: in std_logic_VECTOR(wordSize-1 downto 0);--data in
        dout: out std_logic_VECTOR(wordSize-1 downto 0));--data out
end  ${ram_name};

architecture arch of  ${ram_name} is
  --the memory
  type ram_type is array (0 to cellCount-1) of std_logic_vector(wordSize-1 downto 0);
  signal ram : ram_type := (
    ${ram_data}  -- Memory content to be replaced by list of values
    );
  attribute block_ram : boolean;
  attribute block_ram of RAM : signal is TRUE;
begin
  portIO: process (clk)
  begin
    if (clk'event and clk =  '1' ) then
        dout <= ram(addr);
    end if;
  end process portIO;
end arch;
```