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

Stéphane Mancini

6 novembre 2017

1 Méthodologie

2 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 ont 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.

Chaque étape est une spécialité en soi et l'optimisation générale est assez complexe. 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

- Référence algorithmique
  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.
- Implémentation virgule fixe
  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.
- Implémentation pour la HLS
  En plus de la virgule fixe, les constructions algorithmiques tiennent compte des contraintes de la HLS :
  - Boucles bornées à bornes statiques
  - Pas d'allocation dynamique mémoire et détermination de toutes les mémoires
  - Détermination de toutes les constantes qui seraient des paramètres du code haut niveau (taille des tableaux, etc ...)
  - Interactions avec l'environnement

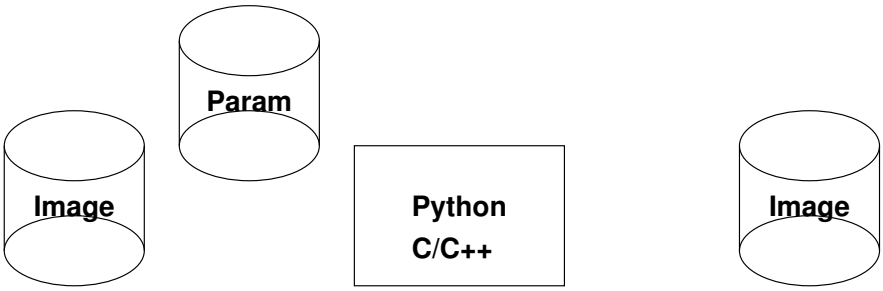

FIGURE 1 – Etape 1 : référence algorithmique

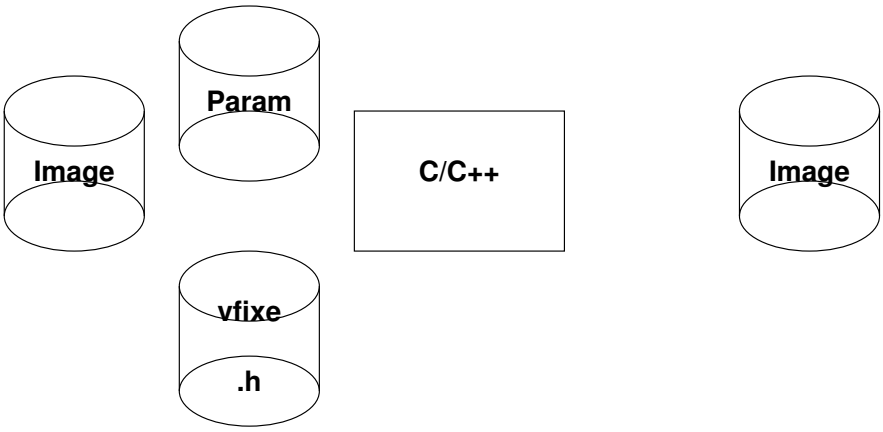

FIGURE 2 – Etape 2 : algorithme virgule fixe

— HLS et réglage de l'architecture
  Premiers tests de HLS
— Vérification post-HLS
  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
— Vérification post synthèse logique
  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

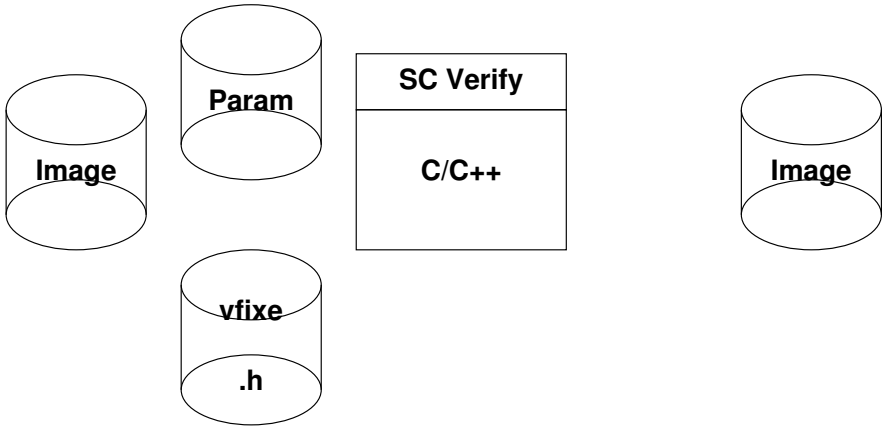

FIGURE 3 – Etape 3 : virgule fixe *pré*-HLS dans l'environnement SC-Verify

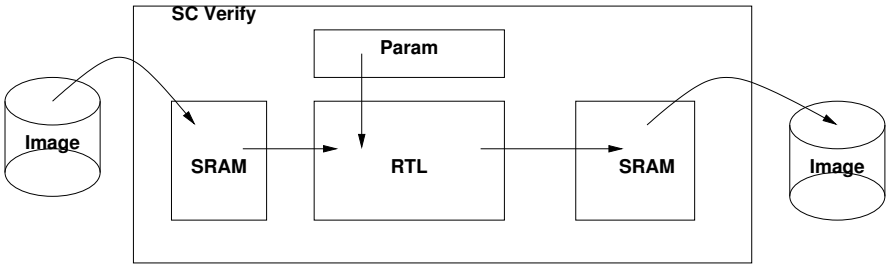

FIGURE 4 – Etape 4 : vérification *post*-HLS dans l'environnement SC-Verify

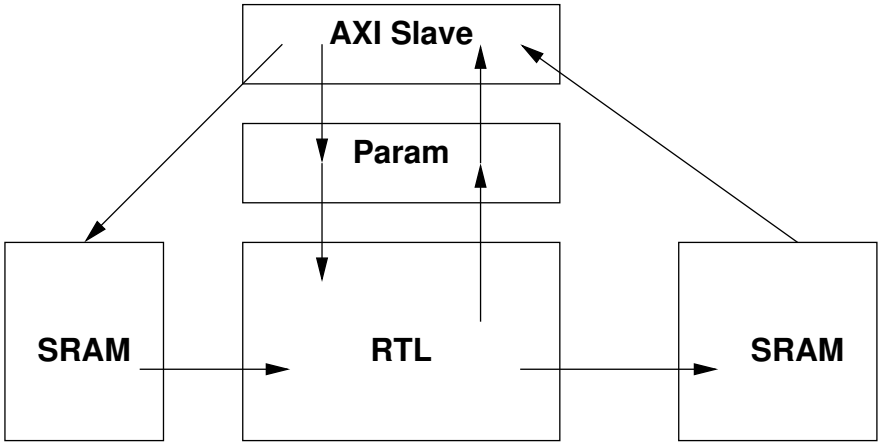

FIGURE 5 – Etape 6 : vérification par émulation dans un système HW/SW ; Les paramètres et les données viennent d'un logiciel (bare-metal par exemple), les résultats sont lus depuis le SW pour vérification.

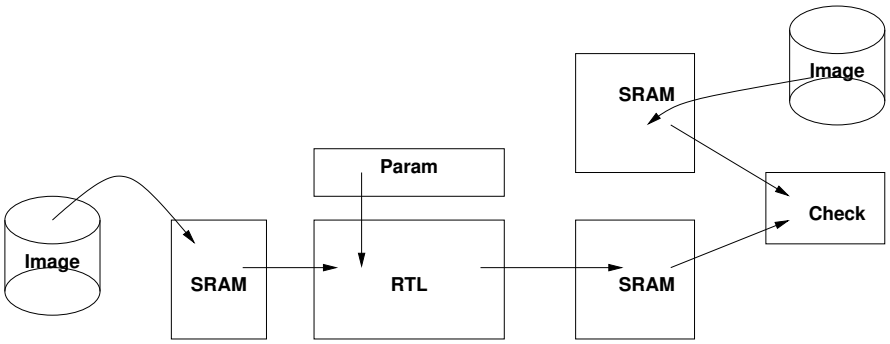

FIGURE 6 – Etape 6 bis : vérification par émulation dans un système HW/SW ; Les paramètres et les données sont statiques, les résultats sont comparés à des résultats attendus.

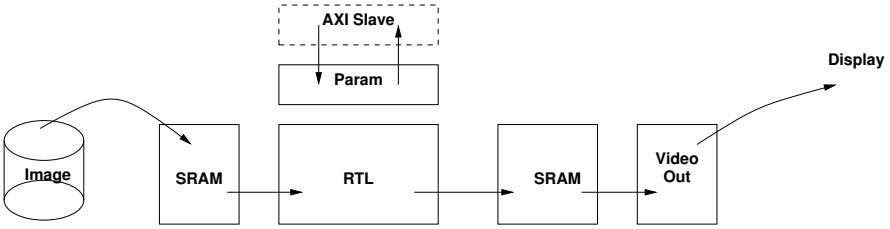

FIGURE 7 – Etape 5 : vérification par émulation ; Les données sont statiques (SRAM initialisée), les paramètres peuvent être statiques ou proviennent d'un SW par le bus AXI, les résultats sont affichés sur écran.

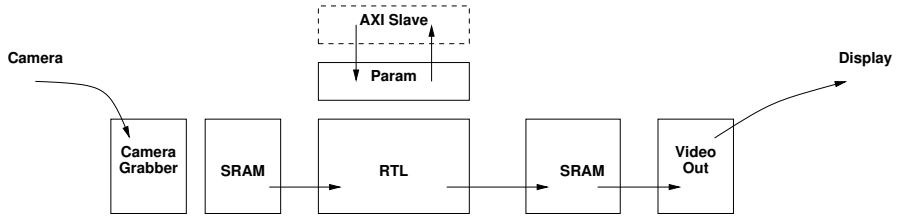

FIGURE 8 – Etape 8 : vérification par émulation ; Les données proviennent d'une caméra, les paramètres peuvent être statiques ou proviennent d'un SW par le bus AXI, les résultats sont affichés sur écran.

2.1 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
- Implémentation virgule fixe et pré-HLS
  Le code pour la HLS utilise la virgule fixe avec les type "AC types" de Mentor graphics. Les paramètres de virgule fixe sont sous forme de macros, regroupés dans un fichier spécifique. L'algorithme est vérifié par l'exécution du code compilé, sur un PC, sans HLS. Les images sont dans es fichiers, les paramètres soit sous forme de macro lorsqu'ils sont statiques, soit en variable globale.
  Il est possible de mesurer l'écart à la référence algorithmique. Les paramètres de la virgule fixe sont assez large et permettent un fonctionnement de l'algorithme.
- 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

  Pour le banc de test, les images d'entrée/sortie sont dans des fichiers
- Vérification post-HLS
  Cette étape est incorporée à la HLS
- 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.
  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.
- 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/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.

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.

3 Images

3.1 Codage des images

Une image est une matrice de pixels, chaque pixel étant soit codés 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.

3.2 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.

3.3 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.

— `PPM` ; Format simple à en-tête (voir `man ppm`), avec les données soit en texte ASCII soit en binaire. 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.

3.4 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.

3.4.1 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++.

3.4.2 En Python

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

4 Accès aux images dans le FPGA

4.1 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 5.1
- 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.

4.2 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 dans l'image, tx doit prendre la taille hor
```

— 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*.

4.3 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.

4.4 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.

5 Annexes

5.1 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
    cellCount : integer := ${ram_cc}; --size of the memory is cellCount*wordSize
    wordSize : integer := ${ram_ws};
```

```
  );
  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 := (
    --   eval TPU_gen_data_bin_lcompute()
    ${ram_data}
    );
  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;
```