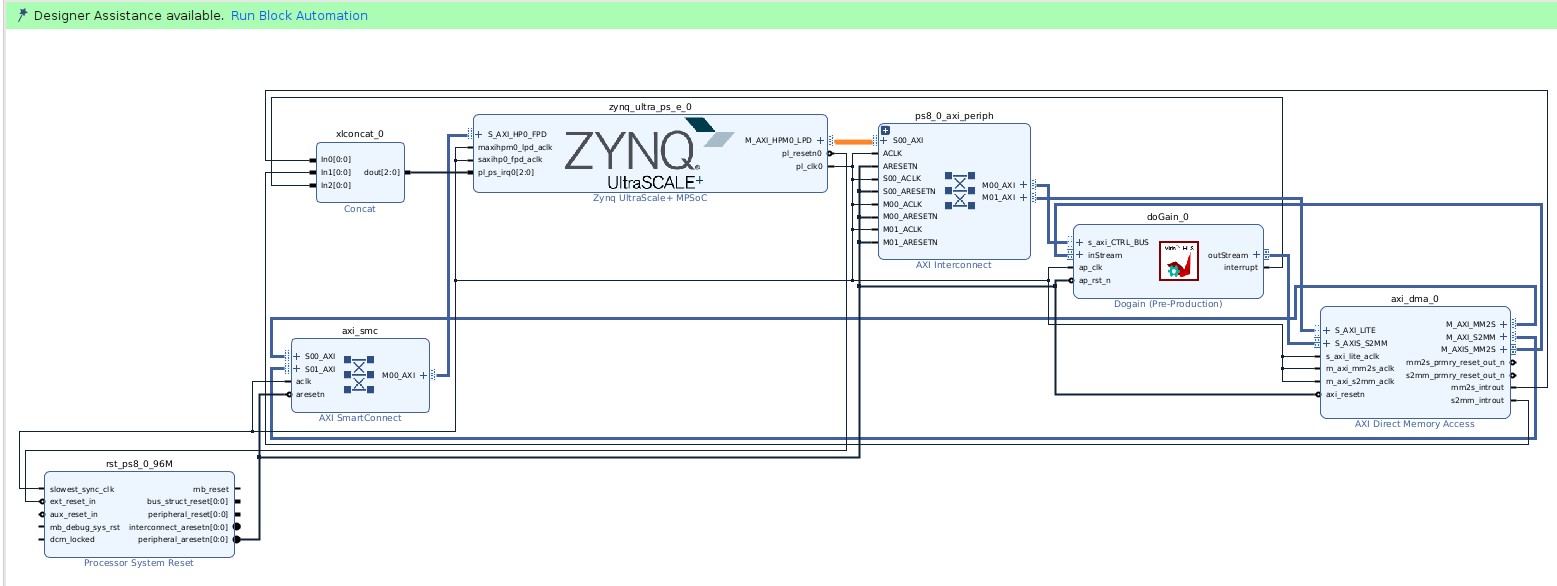
User guide AXI DMA avec IP HLS de type Stream

**Partie Vivado**



L’IP AXI Direct Memory Access est constitué des différentes interfaces suivantes :  
M\_AXI\_MM2S : AXI4 - transfère les données depuis la mémoire du système vers la PL  
M\_AXI\_S2MM : AXI4 - transfère les données depuis la PL vers la mémoire du système  
M\_AXIS\_MM2S : AXI Stream transfère les données de manière continue depuis la mémoire du système vers la PL  
S\_AXIS\_S2MM :  AXI4 - transfère les données de manière continue depuis la PL vers la mémoire du système

L’AXI DMA agit comme un pont entre le domaine AXI4 (utilisé par le processeur et la mémoire système) et le domaine AXI Stream (utilisé par la logique programmable).

On se sert des IPs Processor Reset System et AXI Interconnect qui sont d’usage lors de liaison avec des IPs nécessitant des interfaces de type AXI.

Avec le port M\_AXI\_HPM0\_LPD, le 1 er cœur ARM initie via l’ interface M01\_AXI de AXI Interconnect une transaction avec l’interface S\_AXI\_LITE de l’IP AXI DMA, permettant de configurer des paramètres du DMA tel que la configuration des adresses sources et destinations, la longueur des transferts ect  
  
Le port M\_AXI\_HPM0\_LPD initie également via l’ interface M00\_AXI de AXI Interconnect une transaction avec l’interface S\_AXI\_CTRL\_BUS de l’IP Dogain, permettant d’indiquer le gain qui sera passer en paramètre de la fonction de calcul de l’IP.  
  
Au niveau de la gestion des données par le DMA, celle-ci va être principalement liée aux ports inStream et outStream de l’IP Dogain (contenant un stream).

La prise en charge de ces ports se fera en mode « écriture » avec les interfaces M\_AXI\_MM2S/M\_AXIS\_MM2S, la première ordonne à l’interface slave S\_AXI\_HP0\_FPD de fournir les données contenues dans la DDR que l’on doit envoyer à l’IP Dogain, S\_AXI\_HP0\_FPD répond à la transaction renvoyant celle-ci à M\_AXI\_MM2S, puis M\_AXIS\_MM2S convertie les données en format AXI Stream avant d’effectuer la transaction vers l’interface inStream de Dogain.

Note : le port S\_AXI\_HP0\_FPD est à activer dans la configuration Zynq, il faut pour se faire aller dans PS-PL Configuration -> PS – PL Interfaces -> Slave Interface -> AXI HP, puis sélectionner AXI HP0 FPD.

La prise en charge se fera également en mode « lecture » avec les interfaces S\_AXIS\_S2MM/M\_AXI\_S2MM, la première reçoit le flux de sortie de l’interface outStream de l’IP Dogain, la seconde convertie ce flux du format AXI Stream au format AXI4, et initie une transaction vers l’interface slave S\_AXI\_HP0\_FPD afin d’écrire les données dans la DDR.  
Une fois cela fait, le cœur ARM du Zynq Ultrascale+ lira ces données dans l’emplacement mémoire.

Notons que pour connecter les interfaces maîtres M\_AXI\_S2MM/ M\_AXI\_MM2S à l’interface esclave S\_AXI\_HP0\_FPD, on a fait usage d’un autre IP que AXI Interconnect, AXI SmartConnect (ceci probablement parce qu’il faut un « hub » différent pour un sens de transaction maître esclave différent, le slave étant ici non plus l’IP Dogain mais la PS à travers sa DDR).

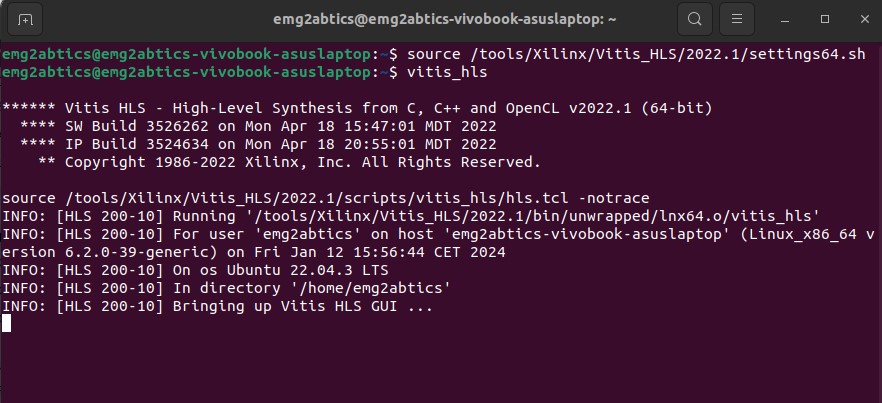
Pour finir avec le DMA, on connectera les signaux horloges liés aux précédentes interfaces à savoir, m\_axi\_mm2s\_aclk, m\_axi\_s2mm\_aclk, s\_axi\_lite\_aclk à l’horloge système pl\_clk0, puis on connectera le signal reset axi\_resetn à peripheral\_aresetn.  
mm2s\_prmy\_reset\_out\_n et s2mm\_prmy\_reset\_out\_n seront laissés déconnectés, mm2s\_introut et s2mm\_introut, signaux d’interruptions de l’axi dma, sont quant à eux concaténés, avec le signal d’interruption interrupt de l’IP Dogain par l’intermédiaire de l’IP Concat, dont la sortie représentative de ces trois signaux d’interruptions, sera amenée au port pl\_ps\_irq du Zynq Ultrascale +.

Note : le port pl\_ps\_irq est à activer dans la configuration du Zynq, il faudra pour se faire aller dans PS-PL configuration -> General -> Interrupts -> PL to PS et mettre IRQ0[0-7] à 1.

**Partie HLS**

Voyons à présent comment créer l’IP Dogain utilisé précédemment avec Vitis HLS

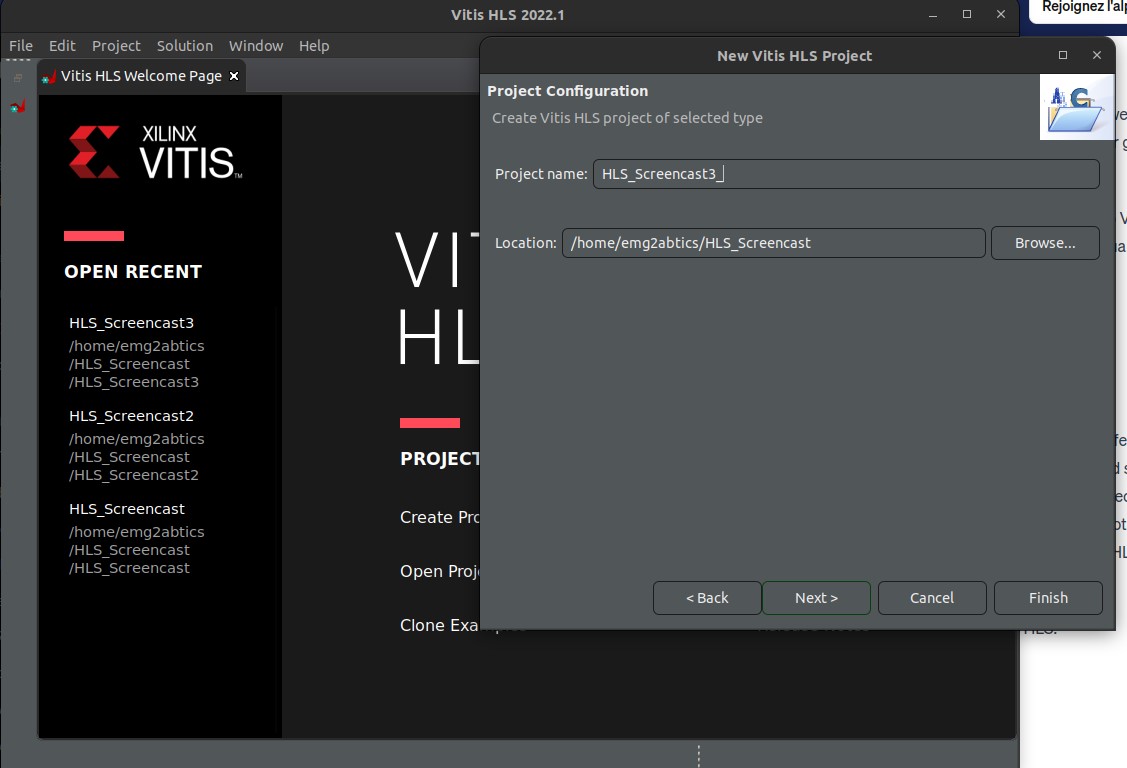
L’IP en question sera de type stream, et effectuera un calcul (ici une multiplication) sur le flux reçu depuis la PS.

On marquera les commandes bash suivantes sous Linux afin de lancer Vitis\_HLS :  
  


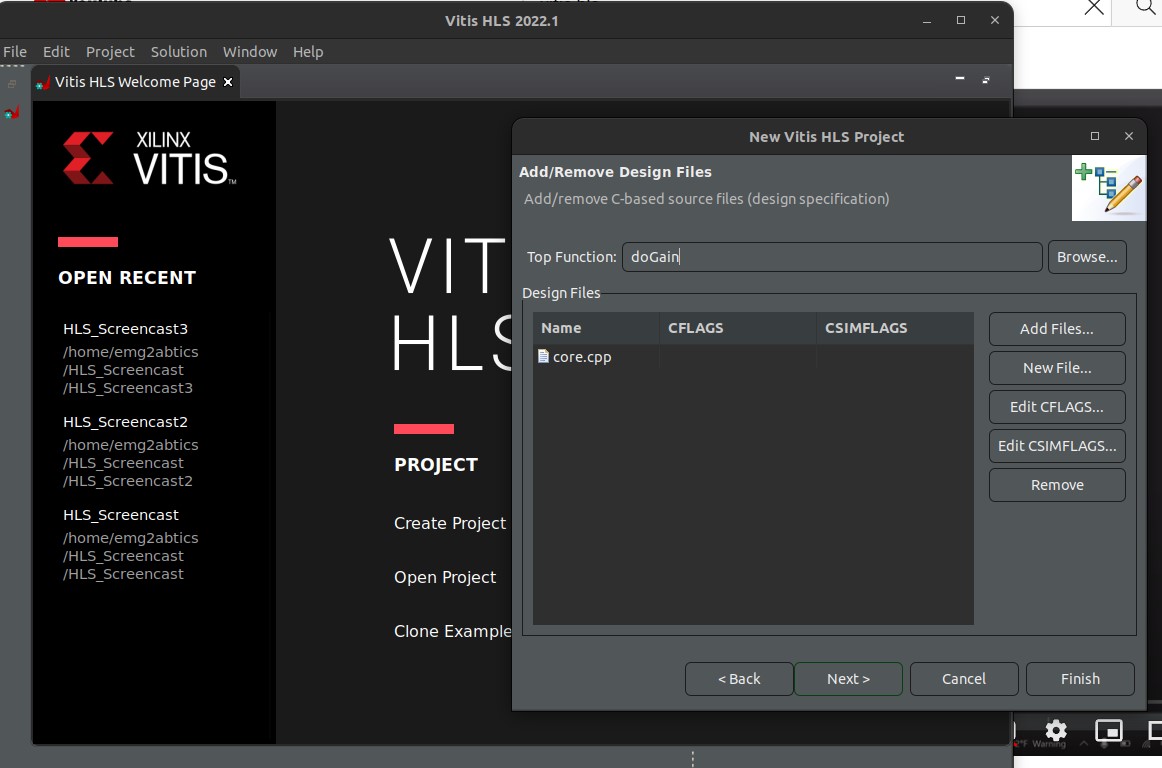
Note : En cas d’initialisation d’un nouvel IDE Vitis\_HLS sous Linux, il est possible que l’application crash quand on la lance, ceci à cause d’une virgule mal placée dans le fichier java chargé de l’exécution du programme.

Se référer alors au topic suivant sur le site d’AMD afin de pouvoir corriger l’erreur dans le fichier de configuration approprié :  
<https://support.xilinx.com/s/question/0D52E00006hpY4PSAU/vitishls-20202-not-starting-only-splash-screen-visible?language=en_US>

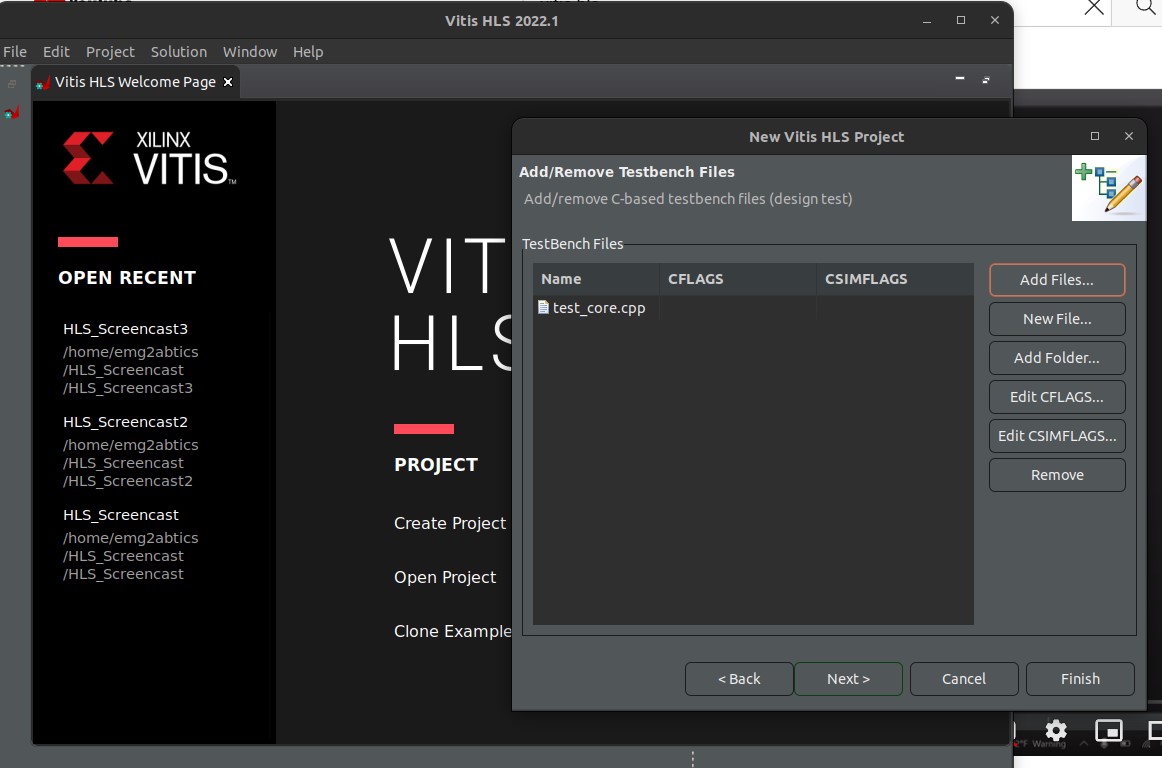
L’IDE Vitis HLS se lance, on clique sur Create Project, puis on précise le nom du dossier du projet, ainsi que son emplacement.



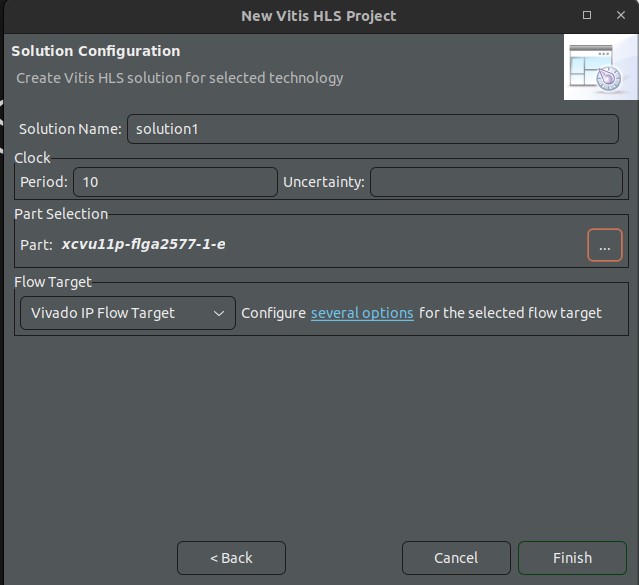
On arrive alors sur la fenêtre permettant de définir la top function, fonction principale qui sera synthétisée par Vitis HLS en RTL pour être utilisable dans un FPGA.  
On ajoute alors core.cpp avec Add Files.



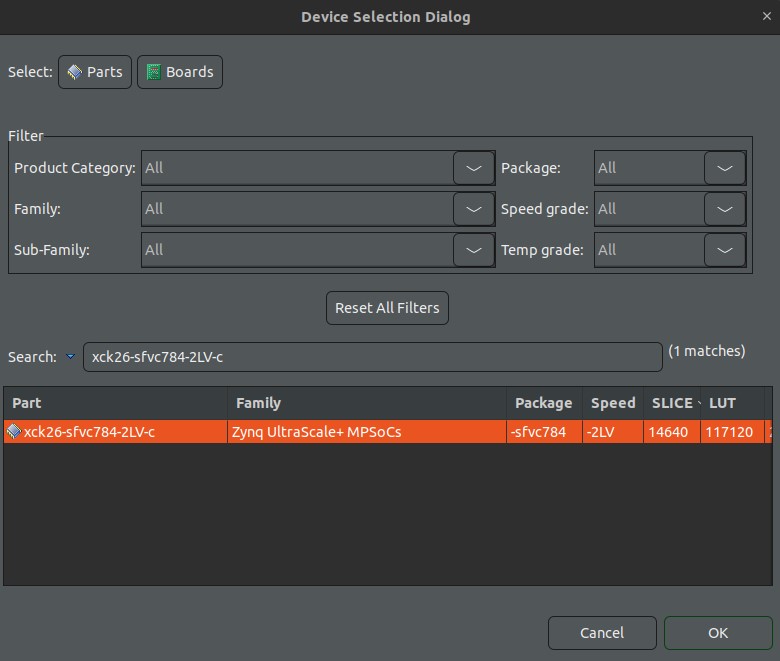
Ensuite, nous sommes sur la fenêtre définissant le testbench associé à la top function.  
On ajoute alors test\_core.cpp avec Add Files.



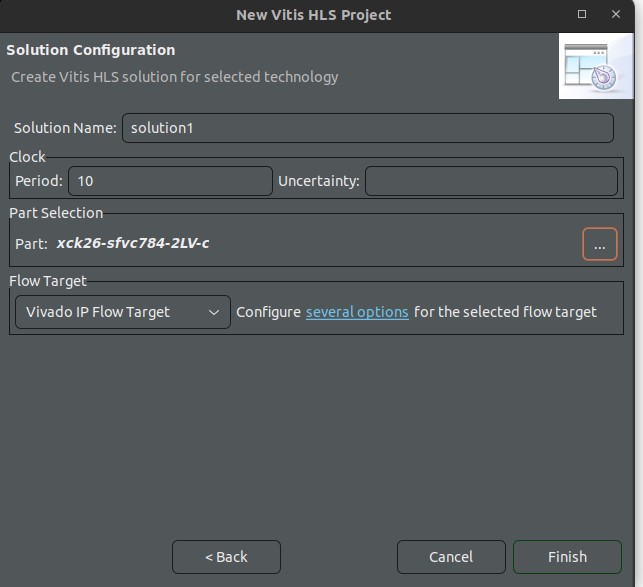
Puis on arrive sur la fenêtre Solution Configuration, permettant de régler des paramètres à appliquer lors de la synthèse du hardware design.  
On ajoutera donc la fréquence d’horloge avec la période à 10, et on précisera l’appareil fpga visé en cliquant sur les 3 petits points.



La fenêtre suivante s’ouvre alors, et nous permet de choisir notre fpga soit avec son part, soit avec le nom de la board.  
On sélectionnera alors le part « xck26-sfvc784-2LV-c » car le part inclus lorsque la sélection se fait par board (ici KV260) est erroné et provoquera une erreur lors de la synthèse.



On retournera alors sur la fenêtre précédente avec notre part chargée, puis on cliquera sur finish pour lancer le projet.



Top function code

la librairie hls::stream est un conteneur générique qui permet d’effectuer une abstraction entre la partie logicielle en C++ et la partie matérielle HLS simplifiant ainsi la gestion des flux qu’il contient.

la librairie ap\_axi\_sdata fournit la classe ap\_axis utilisée pour représenter les signaux AXI Stream, à savoir les données en elles-mêmes ainsi que des signaux de contrôles influant sur divers paramètres liés à celles-ci.

Voici la définition des plus importants des paramètres mentionnés ci-dessus :

TVALID indique que l'émetteur conduit un transfert valide. Un transfert a lieu lorsque TVALID et TREADY sont asserted.

TREADY indique qu'un récepteur peut accepter un transfert.

TDATA est la charge utile principale utilisée pour fournir les données qui transitent à travers l’interface.

TSTRB est le qualificatif d'octet qui indique si le contenu de l'octet associé de TDATA est traité comme un octet de données ou un octet de position.

TKEEP est le qualificatif d'octet qui indique si le contenu de l'octet associé de TDATA est traité dans le cadre des données flux.

TLAST indique la limite (fin) d'un paquet.

TID est un identifiant de flux de données.

TDEST fournit des informations de routage pour le flux de données.

TUSER est une bande latérale d’informations définissant l'utilisateur qui peuvent être transmises le long du flux de données.

Pour des explications plus détaillées sur leur rôle, consultez le lien suivant :  
https://developer.arm.com/documentation/ihi0051/latest/

Lors de l’initialisation de la top function doGain, on définit donc deux conteneurs hls ::stream, inStream et outStream, dont les signaux AXI Stream seront personnalisée avec une structure intSdCh, de type ap\_axis.  
  
On spécifie l’interface avec un pragma HLS INTERFACE axis, afin de fournir une directive au compilateur pour que celui-ci interprète de manière correcte notre objet structuré autour d’une classe ap\_axis (et donc designé en interne pour suivre une structure de donnée cohérente avec le protocole axis).

On spécifie l’interface avec un pragma HLS INTERFACE s\_axilite, afin de fournir une directive au compilateur pour que celui-ci interprète le gain comme étant un paramètre de contrôle pouvant être modifié directement par le CPU, le mot lite est employé car ce genre d’interfaces sont plus légères, nécessitant moins de bus.

Pour ce qui est de return, l’application du pragma HLS INTERFACE s\_axilite est utilisée pour mapper les signaux de contrôle du bloc IP tels que ap\_start, ap\_done, ap\_ready, et ap\_idle dans l'interface s\_axilite.

ap\_start : Contrôle le début du traitement en autorisant le bloc à commencer lorsque le signal est mis à 1.

ap\_ready : Indique la disponibilité du bloc à accepter de nouvelles données en entrée lorsque le signal est mis à 1.

ap\_idle : Signale l'état inactif du bloc, égal à 1 lorsqu'il ne traite pas activement de données.

ap\_done : Indique que le bloc a terminé son opération lorsque le signal est mis à 1.

L’inclusion de bundle=CTRL\_BUS permettra de préciser que l’on parle d’une seule et même interface s\_axilite.

Les paramètres de modèle ap\_axis<32,2,5,6> dans le contexte des éléments AXI Stream sont utilisés de la manière suivante :

32 : Spécifie la largeur du champ de données en bits (TDATA).

2 : Spécifie la largeur du champ utilisateur en bits (TUSER).

5 : Spécifie la largeur du champ d'identifiant en bits (TID).

6 : Spécifie la largeur du champ de destination en bits (TDEST).

De plus, les largeurs des champs strobe et keep sont calculées par Vitis HLS en fonction de la largeur de données spécifiée.

Les métadonnées (T\_\_) se concentrent sur l'individualisation des éléments de données, fournissant des détails spécifiques à chaque élément pour un traitement plus sophistiqué.

Le contrôle au niveau du bloc (ap\_\_) se concentre sur la coordination et la gestion de l'ensemble du bloc, assurant le bon déroulement du traitement global.

On initie une boucle sur 1000 itérations, on lui affectera #pragma HLS PIPELINE afin de paralléliser les tâches et d’optimiser le initiation interval afin de réduire le temps total d’exécution.

On définit deux objets ValIn et ValOut de structure intSdCh, et donc héritant d’une classe ap\_axis, possédant les méthodes appropriées à la gestion des flux.

On affectera alors les signaux lus avec la méthode read() depuis InStream à ValIn.

Le signal data de ValIn sera multiplié par le gain reçu du CPU, puis affecté au signal data correspondant de ValOut.

Pour le reste des signaux de ValIn (keep, strb, user, last, id, dest) seront affectés tel quel à leurs correspondants de ValOut (ce qui permettra de continuer à les faire reconnaître comme des signaux de provenance similaire à ce qu’ils étaient avant modification).

On écrira ensuite dans OutStream les signaux de ValOut à l’aide de la méthode write().

Note : la différence entre le coupe InStream/OutStream et le couple ValIn/ValOut réside dans le fait que le premier concerne des objets ayant une structure intSdCh à l’intérieur d’un conteneur hls::stream, ce qui les disposent particulièrement à être des interfaces d’entrée sortie de l’IP Core performantes, ce qu’elles deviendront en prenant en compte les pragmas axis leur étant affectés (afin d’indiquer au compilateur de les créer en accord avec le protocole axis lors de la synthèse).

Le second couple n’ayant pas de conteneur et de pragmas axis affectés, son rôle reste purement logiciel et non pas matériel.

Test bench de la Top Function

On commence par déclarer le header de notre top function précédemment créé doGain, ayant en paramètre deux conteneur hls::stream InStream et OutStream de structure intSdCh, ainsi qu’un entier gain.

On définit alors la fonction main, ainsi que deux conteneurs hls::stream inputStream et outputStream de structure intSdCh.

On remplit à l’aide d’une boucle inputStream, définissant un objet intermédiaire ValIn de type intSdCh, on affectera à son signal data chaque itération de la boucle.

Pour ses signaux keep, strb et user on affectera 1, pour ses signaux id et dest 0.

Une fois les signaux définis on affecte ValIn à inputStream, obtenant ainsi un conteneur plein.

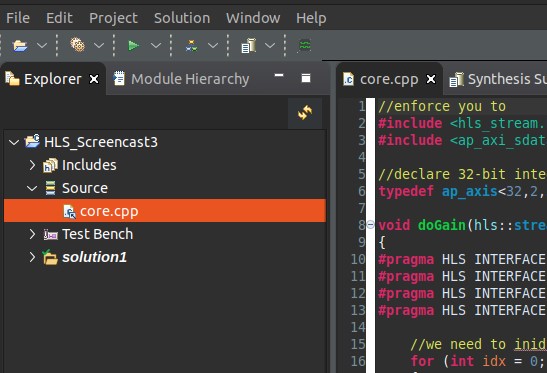
On passe alors en argument de la fonction doGain ce conteneur ainsi que outputstream, en plus d’un entier quelconque (ici 5).

On obtiendra alors un conteneur outputStream ayant un flux avec un signal data modifié par le gain.

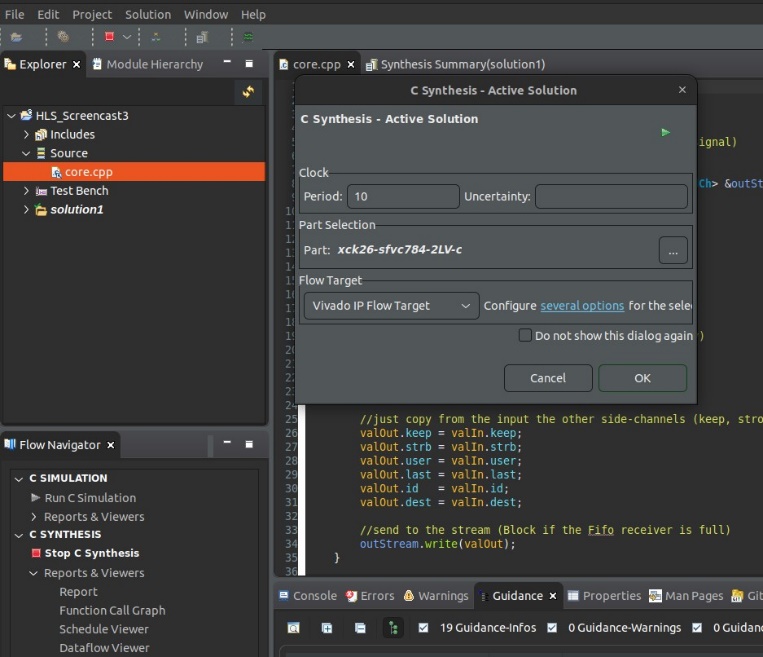
Pour le récupérer, on se servira d’un objet ValOut de type intSdCh, que l’on passera en argument de la méthode read de outputStream.

On affichera alors sur le terminal la valeur du signal data de ValOut avec un printf, en spécifiant static\_cast <int> afin de convertir le type de données renvoyé par axis en entier si ce n’est pas déjà le cas.

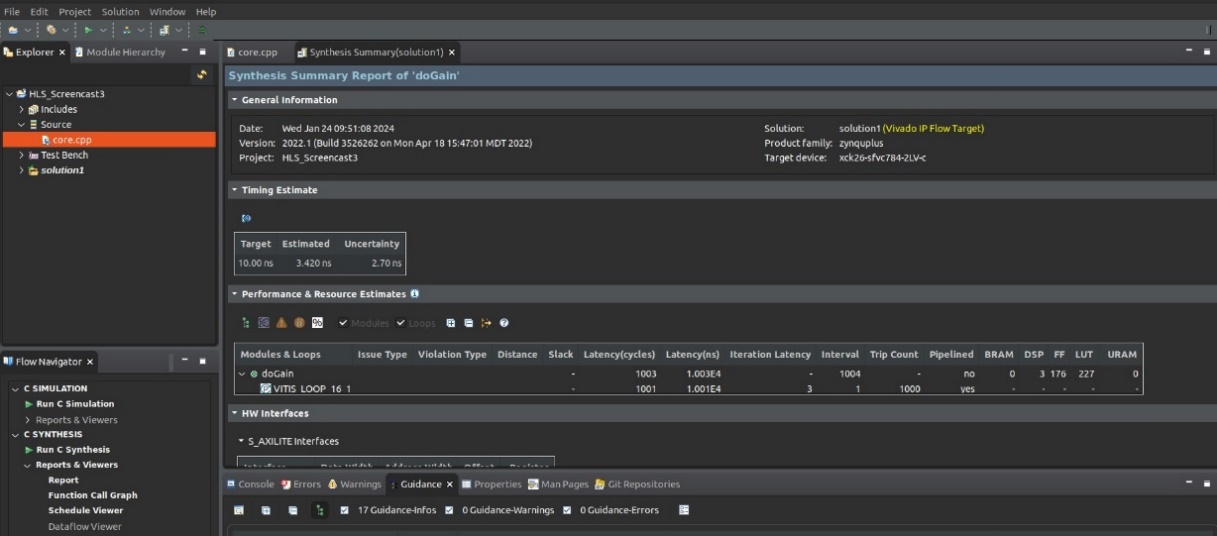
On veut à présent lancer la synthèse du code en RTL/netlist, on clique sur le sélecteur à droite du triangle vert « run flow », puis sélectionne « C synthesis ».  
Celle-ci a pour but d’effectuer une synthèse en RTL/netlist avec une traduction préalable du langage C en VHDL/Verilog.



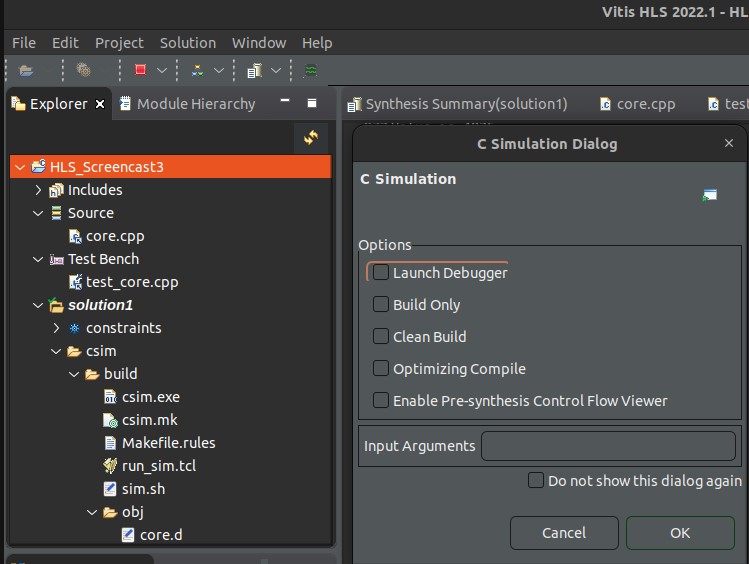
Une fenêtre « active solution » s’ouvre, on la configure avec une période de 10 et la part du som k26.



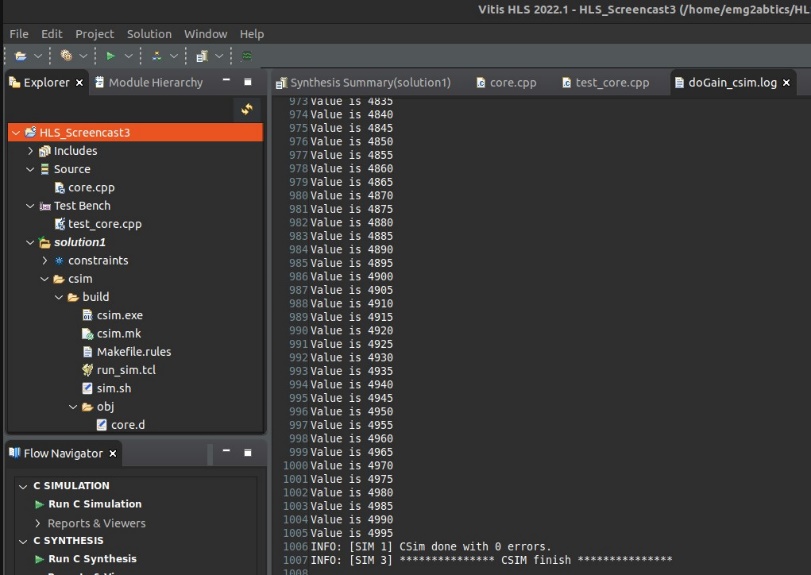
Après avoir valider, on obtient les résultats de la synthèse RTL/netlist de la top function doGain.



On fait ensuite de même pour la fonction du testbench, à ceci près qu’on sélectionne à droite du triangle vert « run flow » non plus « C Synthesis » mais « C Simulation ».  
C Simulation effectuera une simulation utilisant pour vérifier son comportement la top function en conservant celle-ci en C sans appliquer les pragmas.

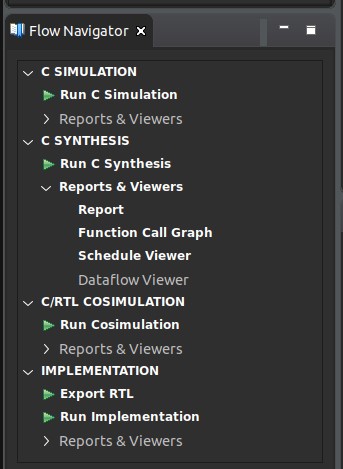


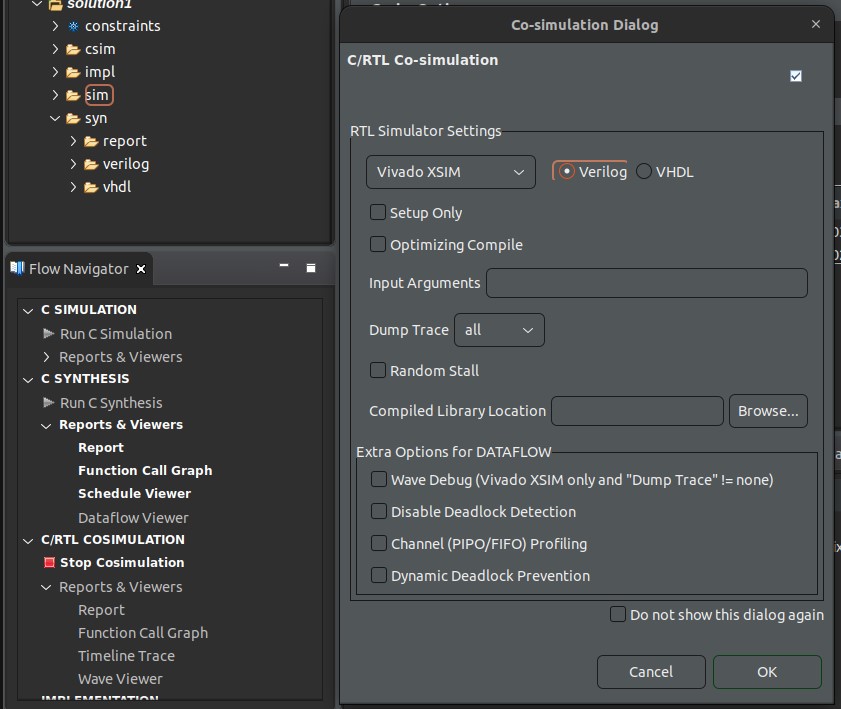
On visualise alors dans un fichier .log s’ouvrant les résultats du printf, montrant le flux en sortie de la fonction doGain.



Dans le flow navigator section C /RTL COSIMULATION, on clique sur Run Cosimulation.

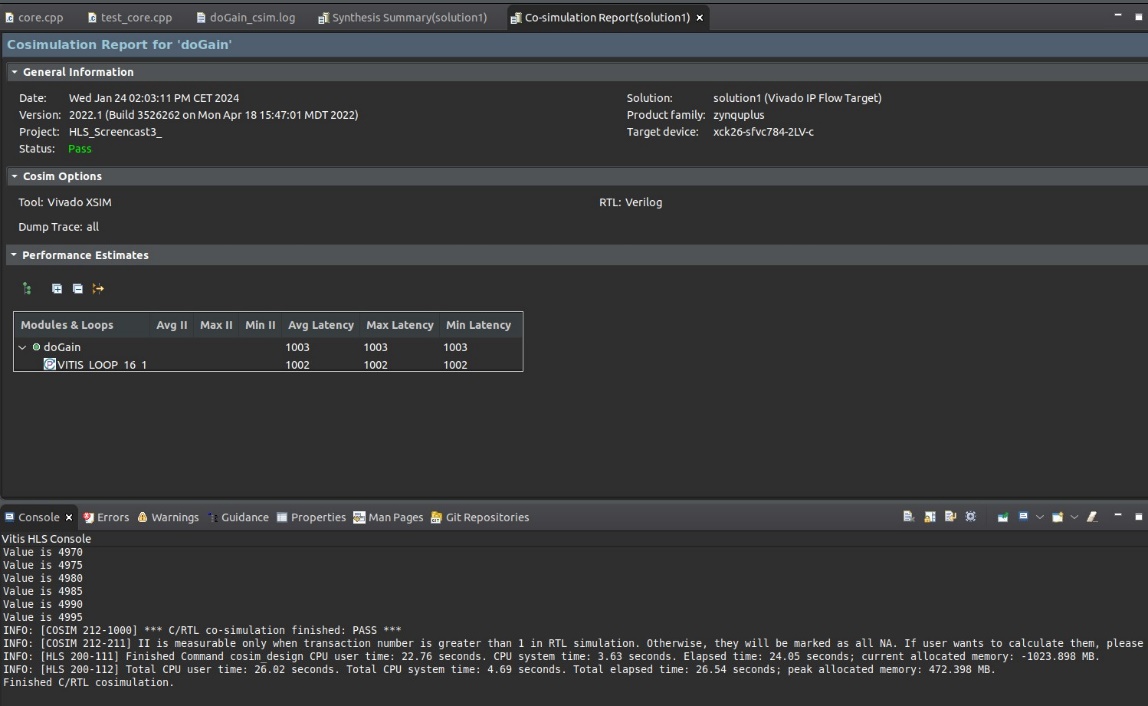
La co-simulation effectuera une simulation similaire à la précédente en C, à laquelle sera rajoutée une simulation basée sur les résultats de la synthèse précédemment effectuée, avec donc une logique RTL, ceci afin de vérifier que les performances du code VHDL généré sont proches voir meilleures que celles du code original en C (et donc que l’optimisation est correcte).  
Une fois que les résultats de la co-simulation seront obtenus, on visualisera ses signaux dans un outil appelée XSIM, que l’on ouvre avec le bouton Open Wave Viewer.



On commence par ouvrir la fenêtre de configuration de la co-simulation :  
  
  
  
On paramètre la fenêtre de co-simulation avec l’outil de simulation de signaux par défaut Vivado XSIM, on sélectionne Verilog comme langage HDL, et on met Dump Trace sur all.

Pour le choix de Verilog, celui-ci est primordiale en l’état de mes connaissances actuelles, car les IDE Xilinx utilisent l’ancienne version de VHDL, et celle-ci ne permet de choisir quand stopper la simulation, en effet elle s’arrête automatiquement quand il n’y a plus de code.  
Cela rend l’accès à Open Wave Viewer impossible, cependant Verilog n’a pas ce genre de limitation et permet donc un accès à Open Wave Viewer.

On aura alors après la co-simulation effectuée l’affichage du rapport.

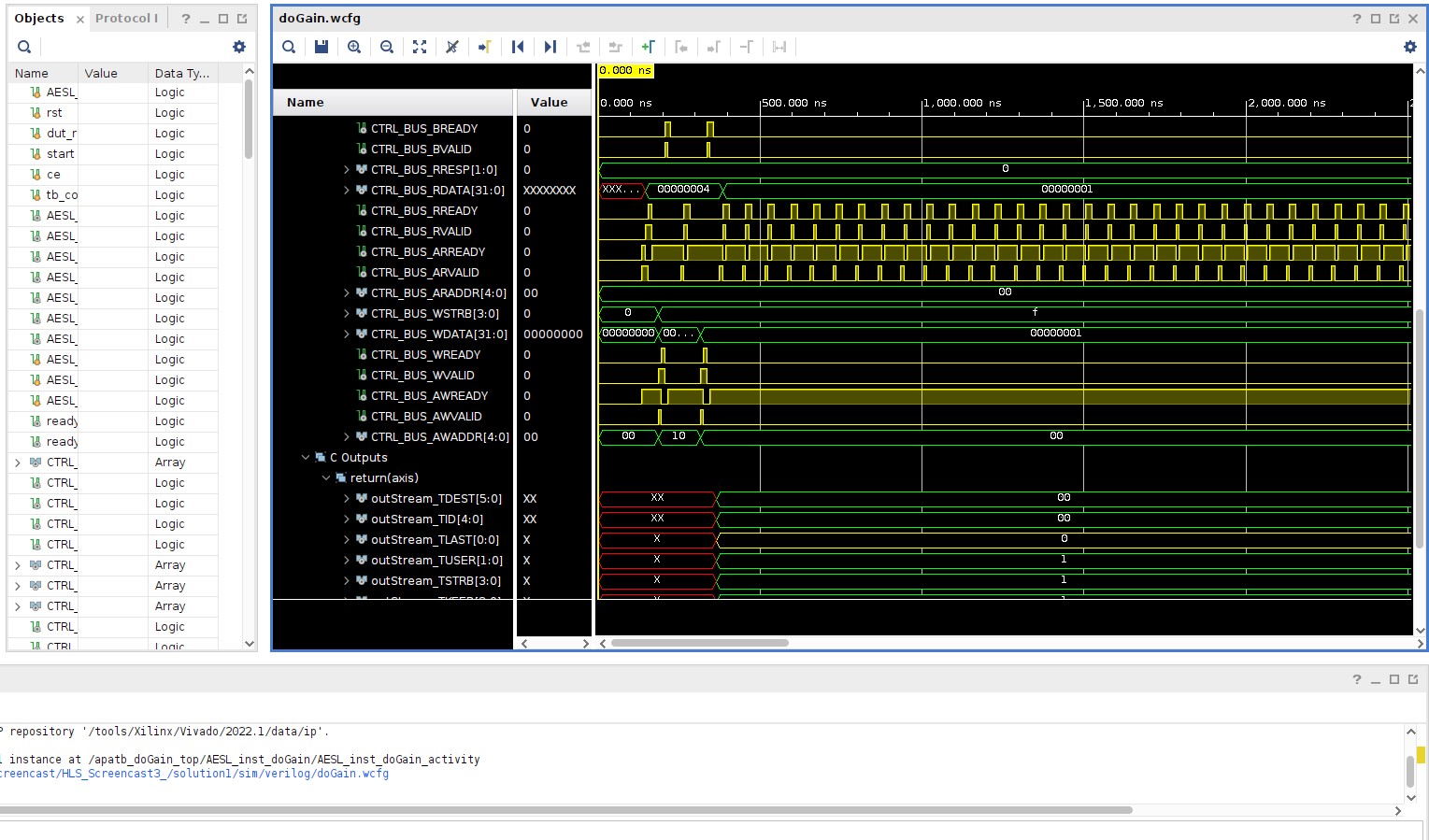


Note : Lors de la co-simulation, on a eu une erreur *« WARNING: [XSIM 43-3431] One or more environment variables have been detected which affect the operation of the C compiler »*, déclenchée par une mauvaise utilisation de la variable LIBRARY\_PATH.  
Cette erreur provoquait une non visibilité des signaux une fois le wave viewer ouvert sous Vivado.  
Pour corriger cela, on a donc écrit la commande suivante dans le terminal ubuntu « export LIBRARY\_PATH=/usr/lib/x86\_64-linux-gnu ».  
L’erreur sera toujours affichée mais les signaux apparaîtrons quand XSIM s’ouvrira dans Vivado.

Note \*\* : Le fait que ce problème arrive de manière régulière n’est pas attestée, il se peut que les signaux apparaissent sans problème, dans ce cas sautez cette étape.

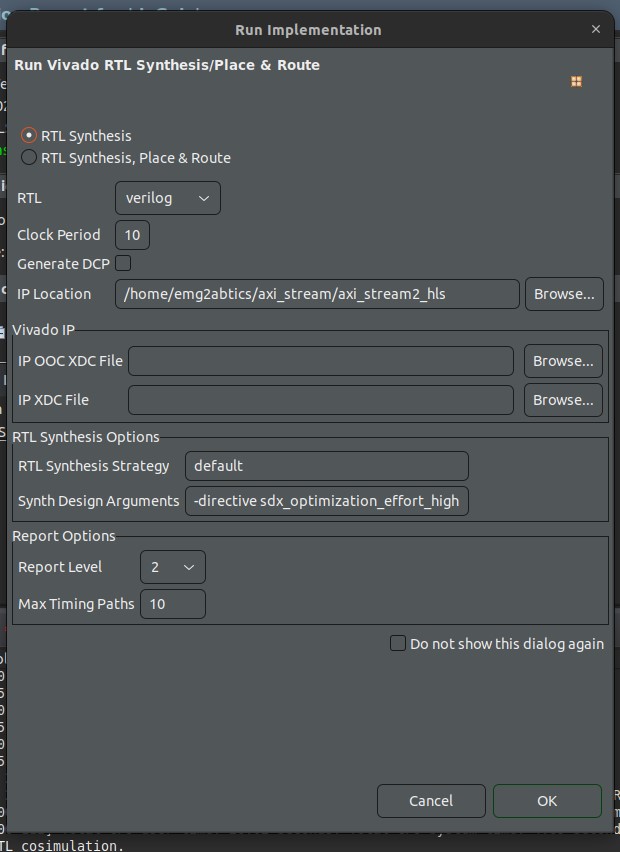
On appuie ensuite sur « open wave viewer » que l’on trouve dans Solution.

Une fenêtre Vivado s’ouvre alors, nous permettant de visualiser nos signaux importants tel que les side channel ou les signaux de contrôle du bloc.

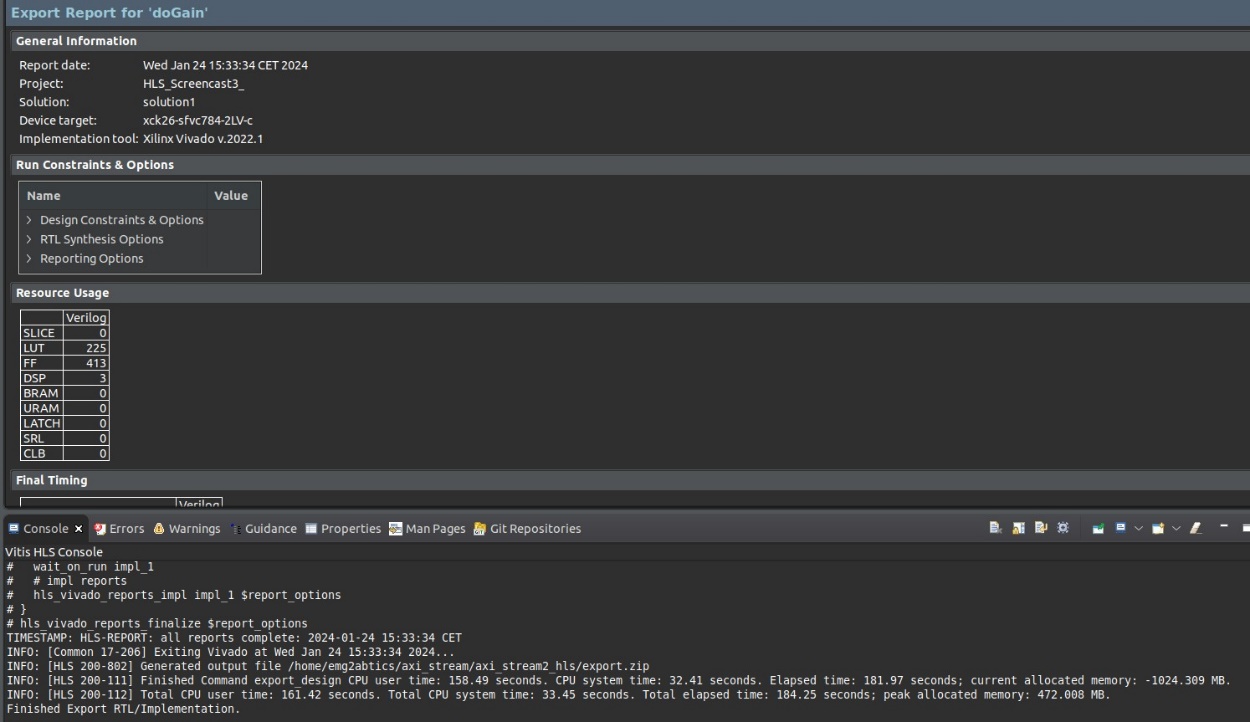


De retour dans Vitis HLS, dans le flow navigator section IMPLEMENTATION, on clique sur Run Implementation, afin de générer le bitstream.

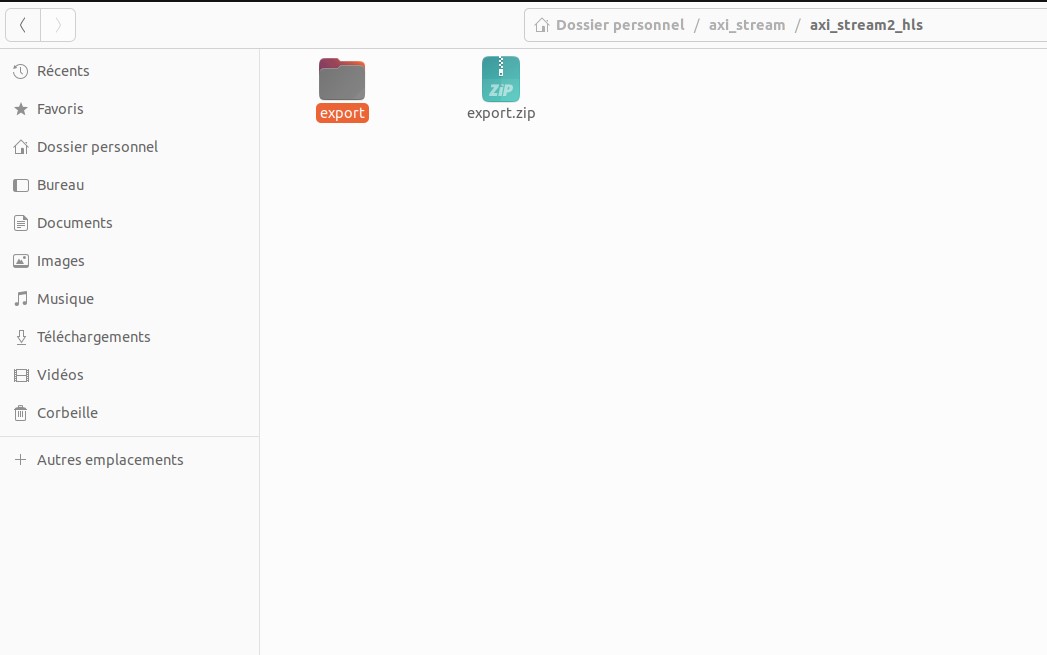
Une fenêtre de configuration s’ouvre, on sélectionne RTL Synthesis, Verilog comme RTL, une clock period à 10 et on précise l’emplacement dans lequel on veut stocker le bloc IP une fois celui-ci généré.



On aura ensuite le rapport d’implémentation s’affichant :



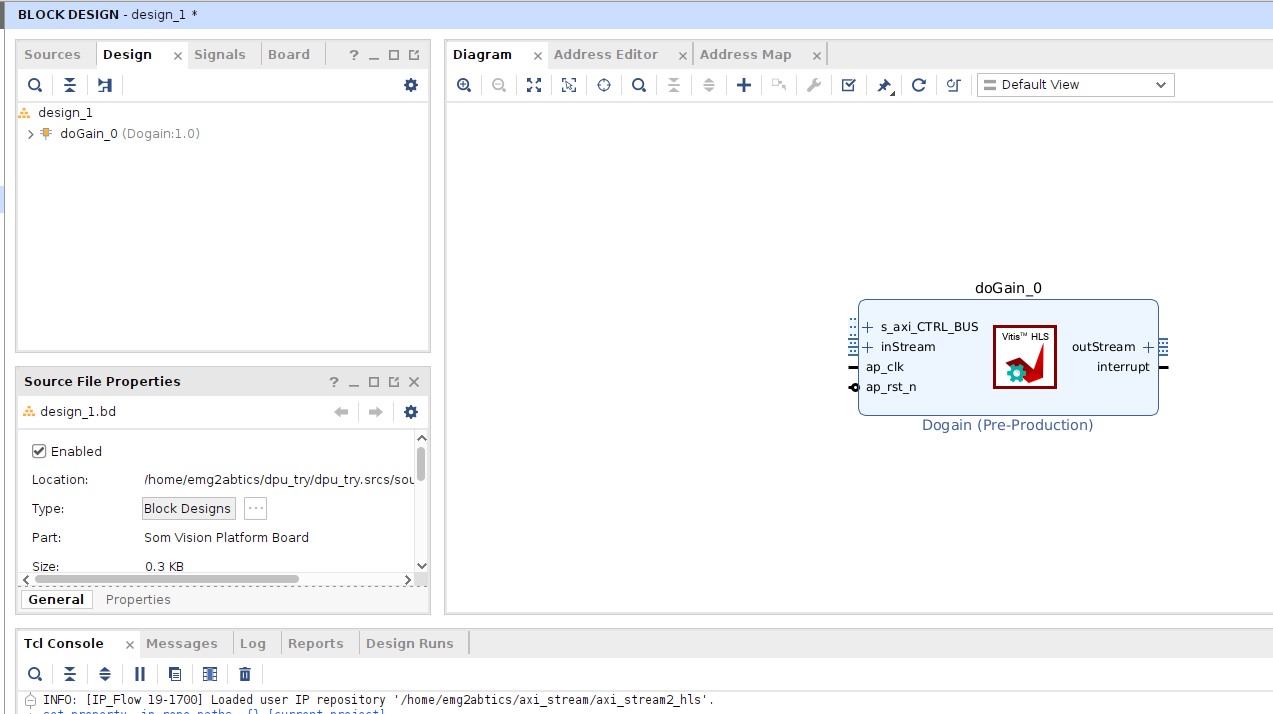
On va chercher après cela le .zip qui a été créé lors de l’implémentation/exportation dans le chemin précédemment spécifié. On l’extraira alors, afin d’avoir accès au dossier « export ».



On va alors sur Vivado, et après avoir ouvert un projet, on clique sur Settings dans le Flow Navigator, puis on va dans la section IP, et on clique sur Repository, on précisera lors le chemin jusqu’au dossier « export », l’IDE Vivado sera capable de détecter l’IP HLS y étant contenu, IP qu’on inclura alors dans la bibliothèque d’IPs connus en cliquant sur ok.

Note : C’est également pour cette raison que l’on a dézippé le .zip, car Vivado n’est pas capable autrement de repérer l’IP HLS.

On ajoutera alors l’IP en question au projet avec le bouton « + », afin de l’intégrer au reste du projet.



**Partie Vitis**

On définit la fonction initPeripherals ayant pour but d’initialiser l’IP Core et le DMA.

On récupère la configuration de doGain par le résultat de la fonction XDogain\_LookupConfig (à qui on passe l’ID de doGain en paramètre).

On applique en paramètre de la fonction XDogain\_CfgInitialize cette configuration, ainsi que l’adresse de l’instance doGain.

On vérifie alors le statut du résultat retourné pour s’assurer que l’initialisation a été un succès.

On récupère la configuration de axiDma par le résultat de la fonction XAxiDma\_LookupConfig (à qui on passe l’ID de axiDma en paramètre).

On applique en paramètre de la fonction XAxiDma\_CfgInitialize cette configuration, ainsi que l’adresse de l’instance axiDma.

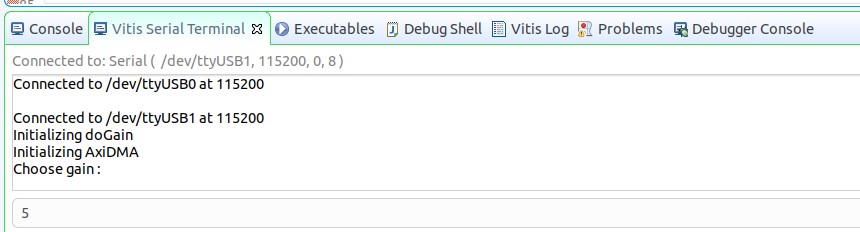
On vérifie alors le statut du résultat retourné pour s’assurer que l’initialisation a été un succès.

On désactive alors les interruptions possibles avec la fonction XAxiDma\_IntrDisable, à la fois dans le sens de l’IP Core vers le DMA ainsi que dans le sens du DMA vers l’IP Core.

On entre alors dans la fonction main, on commence par initialiser les adresses des buffers de réception et de transmission.

On exécute ensuite la fonction intiPeripherals définis précédemment, puis on remplit un tableau d’entier inStreamData avec 1000 valeurs.

On entre dans la boucle infinie, on laisse l’utilisateur définir le gain dans le terminal avec la fonction scanf, celui-ci sera affectée à la variable gain de type int.



Cette même variable est ensuite passée à la fonction XDogain\_Set\_gain, qui utilise des drivers pour piloter l’interface CTRL\_BUS et définir le gain interne à l’IP Core.

On déclenche ensuite le signal de contrôle ap\_start de l’IP Core à l’aide de XDogain\_Start.

On flush alors avec la fonction Xil\_DCacheFlushRange les buffers de transmission (inStreamData) et réception (m\_dma\_buffer\_RX) afin que les valeurs du flux (avant et après modification du gain par l’IP Core) qui sont rangées dans le cache parce que variable locale, soient écrites dans la zone mémoire de la DDR leur correspondant.

Ensuite, on effectue les transferts des données maintenant disponibles dans la DDR à l’aide de la fonction XAxiDma\_SimpleTransfer, en transmettant inStreamData dans le sens DMA to DEVICE, puis en recevant les données dans le sens DEVICE to DMA dans m\_dma\_buffer\_RX.

On attend alors avec la fonction XAxiDma\_Busy dans une boucle while que le axiDma ne soit plus occupé dans un transfert DEVICE to DMA.

Puis on invalide le cache lié à m\_dma\_buffer\_RX avec la fonction Xil\_DCacheInvalidateRange, pour éviter que le CPU lise des données obsolètes provenant d’un précédent transfert du DEVICE vers le DMA, mais bien les données actuelles stockées dans l’emplacement DDR correspondant.

Avec la fonction XDogain\_IsDone dans une boucle while, on attend que le signal de contrôle ap\_done de l’IP Core soit à 1, ce qui signifie que le transfert des données est fini.

On affiche alors à l’aide d’un printf dans une boucle for toutes les données reçues dans m\_dma\_buffer\_RX dans le terminal.

