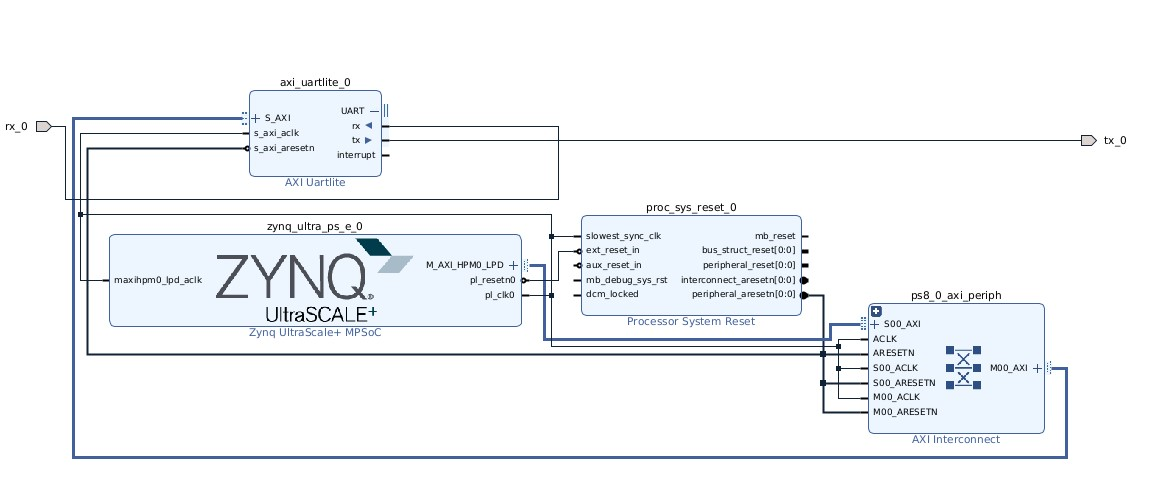
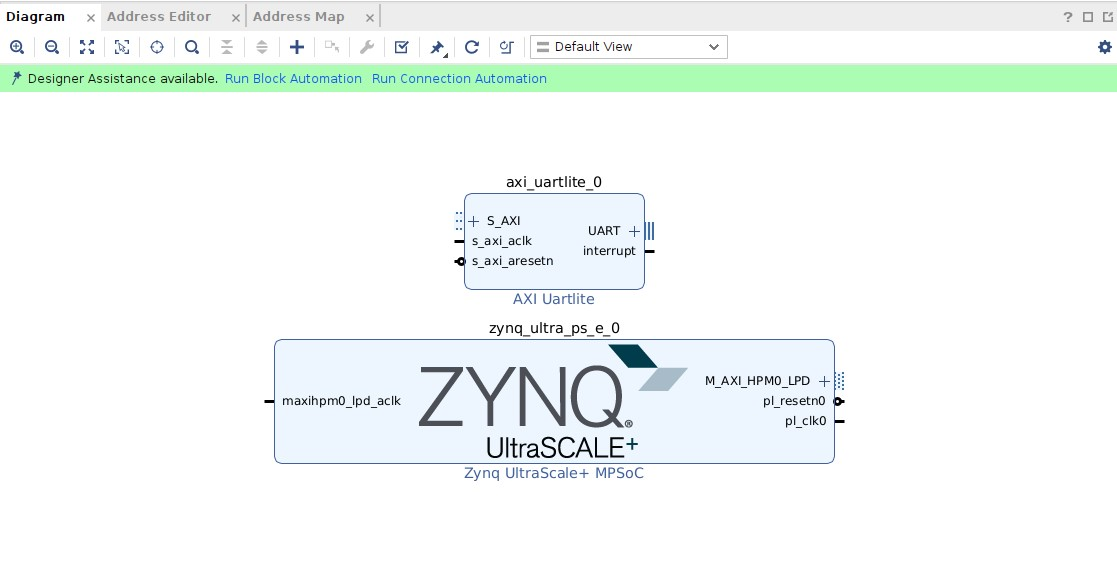
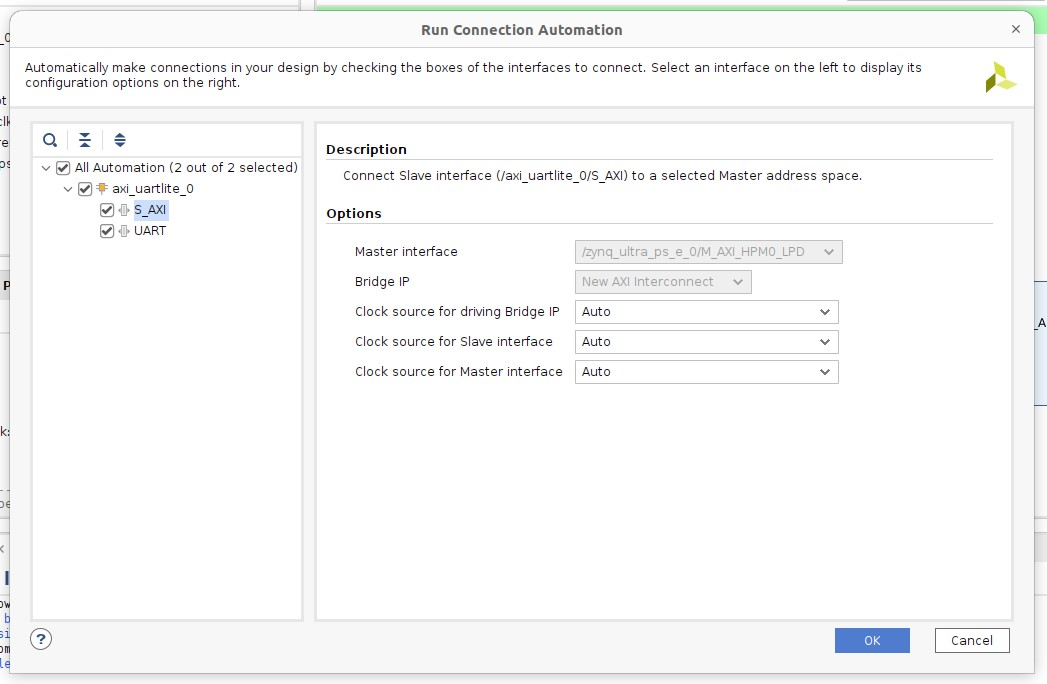
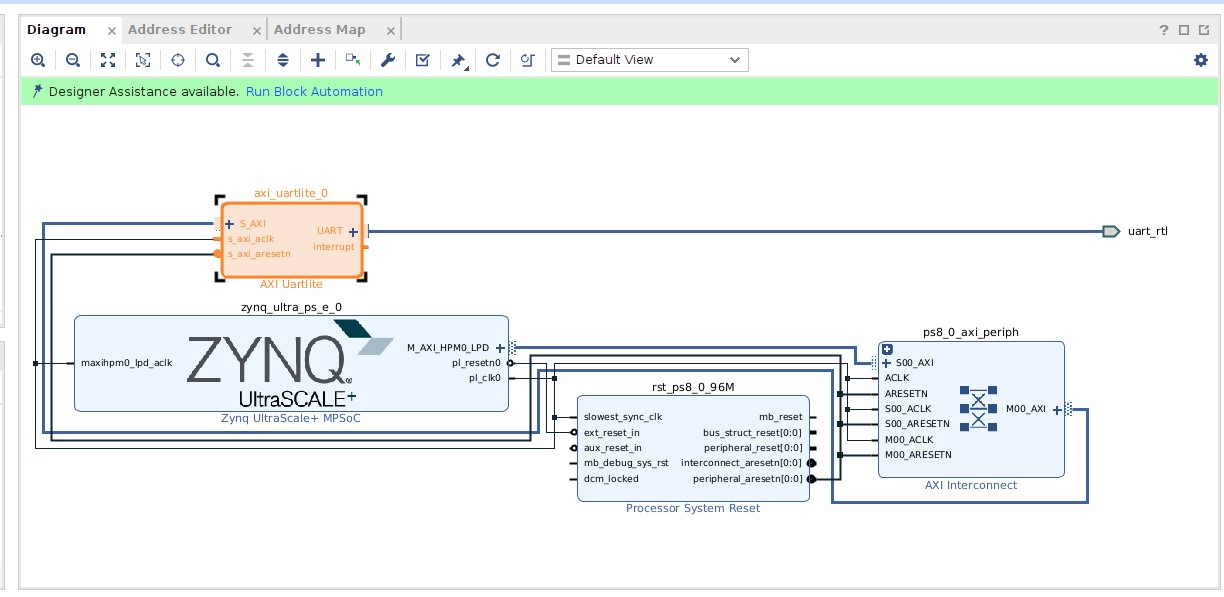
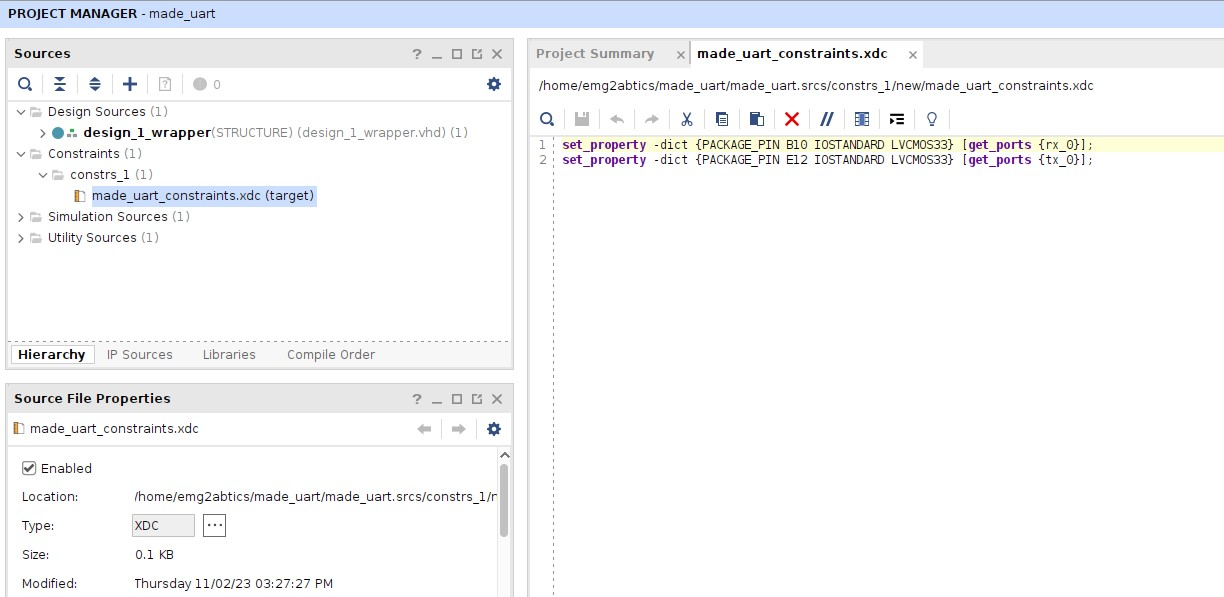
**User guide Uart Lite ABTICS**

On ouvre Vivado et on crée un projet comme dans le précédent user guide.  
On ajoutera l’IP Zynq Ultrascale + MP SoC (représentant le processing system) ainsi que l’IP Processor System Reset, servant à rendre le signal reset utilisable par des IPs ayant une communication de type AXI.  
Avec l’IP AXI interconnect vu dans le précédent user guide, on relie le port « maître » M\_AXI\_HPMO\_LPD (signal commandé par le premier cœur ARM) de l’IP Zynq Ultrascale + MP SoC au port « esclave » S\_AXI de l’IP AXI Uartlite.  
L’IP AXI Uartlite est un UART (composant matériel utilisé pour la communication asynchrone entre dispositifs électronique) créer dans la partie PL (Programmable Logic).  
Les signaux clock et reset à acheminer pour le bon fonctionnement des IPs utilisant une communication AXI se fera comme indiqué dans le précédent user guide.  
Le Zynq pourra alors initier des transactions, à savoir ici des opérations de lecture et d’écriture auxquelles l’ AXI Uartlite répondra.



A noter qu’il est possible pour des schémas simples, de mettre un IP initial (ici le zynq) et final (ici uart lite), puis de sélectionner run automation.  
  


On aura alors l’écran suivant, avec le type de « connection » à effectuer, ici connecter l’interface esclave S\_AXI de l’IP Uartlite à l’interface « maître » de l’IP Zynq Ultrascale + MP SoC.  
  


On aura alors les IPs intermédiaires nécessaires à la mise en place de cette « connection » qui seront automatiquement créés par Vivado (à savoir Processor System Reset et AXI Interconnect), ainsi que le routage approprié des interfaces (notamment les clock et reset).  
  
  
  
Néanmoins, si on reprend le premier diagramme fait à la main sans « Run Block Automation », on remarque qu’au lieu d’un label unique appliqué à une interface UART de type AXI, on a choisi de créer un label pour rx et tx, ceci afin de pouvoir personnaliser le fichier de contraintes, avec un pin dédié à la réception et un autre à la transmission.  
  
  
  
On alors avec les PIN PACKAGE suivants le port rx définit sur le 1er pin inférieur du PMOD, et le port tx définit sur le 2ème pin inférieur du PMOD.  
  
Pour vérifier plus en détail les correspondances PIN PACKAGE et positions réelles sur la carte KV260, regardez le premier user guide, où une explication complète est fourni.  
  
On clique ensuite sur generate bitstream dans le flow navigator (ce qui générera également les étapes précédentes de synthèse et d’implémentation).

Après avoir exporter le .xsa dans le dossier du projet vivado, à l’aide de file -> export -> export hardware, on peut ensuite lancer vitis en cliquant sur tools -> Launch Vitis IDE.  
  
A noter qu’il faudra spécifier l’emplacement du worskpace sur lequel on souhaite travailler (en général on laisse l’emplacement du projet vivado comme chemin).  
  
Attention, pour ce projet particulier, le workspace n’est pas juste l’emplacement du projet à savoir made\_uart, il faut aller dans un sous dossier de celui-ci à savoir « f1 ».  
  
Une fois sous Vitis, comme préciser dans le user guide précédent, on crée un « application project », en sélectionnant le .xsa exporter précédemment comme étant notre BSP.  
On sélectionnera un type de projet par défaut (ici « hello world »).  
  
Bien qu’ayant la possibilité de se servir d’exemples standards mise à disposition par Xilinx pour ses IPs (il faut cliquer pour ce faire dans le fichier plateform.spr du dossier design\_1\_wrapper, qui symbolise notre BSP), on a ici choisi d’effectuer un code personnalisé afin d’une part, d’éviter le fonctionnement trop lourd des fonctions XUARTLite\_Send et XUARTLite\_Receive, qui renvoyait plusieurs fois la donnée à envoyer, et d’autre part, d’adapter le code à notre configuration spécifique, car n’ayant pas de convertisseur uart - micro usb fonctionnel au moment de la réalisation du projet, on a opté pour un rebouclage manuel de tx sur rx afin de vérifier le bon fonctionnement de l’uart lite.

Pour ce qui est de la logique du code (écrit dans helloworld.c dans l’application f1\_1\_app), on commence par créer une instance de classe XUARTLite.

On écrit deux fonctions, send\_byte et receive\_byte.

Pour send\_byte, on précise simplement la fonction XUARTLite\_SendByte, avec ses deux arguments que sont l’adresse de l’IP à savoir XPAR\_UARTLITE\_0\_BASEADDR disponible dans la librairie xparameters, ainsi que la data à envoyer (ici fourni en paramètre de la fonction, en sachant que la taille de celle-ci est d’un octet « u8 »).

Pour receive\_byte, on crée une variable, received\_data de taille u8, que l’on passera en argument de la fonction XUARTLite\_RecvByte, ainsi que l’adresse de de l’IP UartLite, ceci afin de récupérer une information sous forme d’octet et de la stocker dans received\_data.

Dans la fonction main, on définira la variable à envoyer comme contenant 0x03.  
On utilisera alors les fonctions susnommées afin d’envoyer et recevoir la data, puis on l’affichera à l’aide dun xil\_printf sous forme d’entier dans le terminal.  
  
A noter qu’il faut activer le terminal, en cliquant sur windows, show view, puis en écrivant vitis serial terminal, il faudra aussi configurer celui-ci avec le premier ttyUSB s’affichant une fois qu’on a cliqué sur +.

A noter que la fonction usleep fonctionne en micro secondes, le délai entre l’envoi et le début de la réception sera donc d’au moins 0.1 seconde.

