École Polytechnique de Montréal

Laboratoire #2 : Routeur sur puce FPGA

INF3610 – Automne 2015

Arnaud Desaulty
Frédéric Fortier

Département de Génie Informatique

1. Objectif

L'objectif principal de ce laboratoire est de concevoir une application temps réel pour un système embarqué en ayant recours au RTOS µC et de l'exécuter sur un processeur ARM implanté sur une puce FPGA.

Plus précisément, les objectifs spécifiques du laboratoire sont:

- Approfondir ses connaissances de µC.
- S'initier aux environnements de développement de SoC, tel Xilinx ISE.
- Étudier les spécificités de la programmation pour SoC.
- Se familiariser avec les architectures multiprocesseurs asymétriques (AMP)
- Utiliser et comprendre un mécanisme d'interface entre modules logiciels et matériels.

2. Mise en contexte

La ***figure 1*** illustre un réseau téléinformatique permettant l'échange de paquets d'une source à une destination. Dépendamment de la destination, les paquets transitent à travers un ou plusieurs routeurs.

Par exemple, pour aller de la ***source 0*** à la ***destination 1***, les paquets vont passer par le routeur 1, le routeur 2 et le routeur 4 alors que pour aller de la ***source 0*** à la ***destination 2***, les paquets vont passer par le routeur 1, le routeur 3 et le routeur 5. Le routeur 1 devra donc regarder l'adresse de destination de chaque paquet pour décider si ce dernier doit transiger vers le routeur 2 ou le routeur 3. La fonction principale d'un routeur est donc de prendre un paquet et de le renvoyer au bon endroit en fonction de la destination finale. Il devra du même coup rejeter les paquets qui ne sont pas dans son espace d'adressage. Une deuxième fonction est de vérifier les erreurs de transmission à travers un calcul de type CRC (***checksum***). Finalement, un routeur peut aussi supporter une qualité de service (***QoS***) en triant les paquets selon qu'il s'agit par exemple d'un paquet audio, vidéo ou encore contenant des données quelconques.

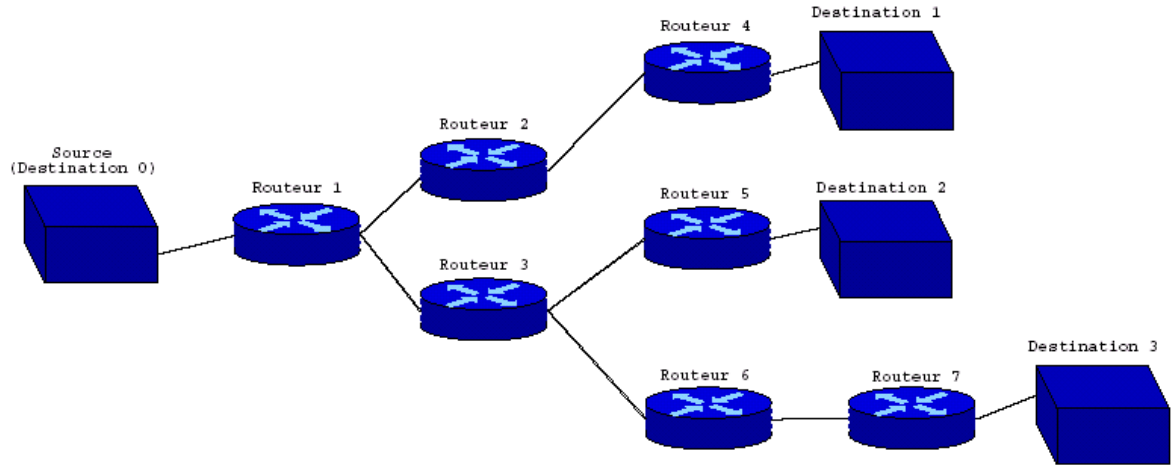


Figure 1 Exemple de Réseau téléinformatique

Évidemment, un routeur peut supporter bien d'autres fonctions. Toutefois, dans ce laboratoire, nous nous concentrerons sur les trois énumérées au paragraphe précédent. Le format des paquets sur le réseau est le suivant :

4 Octets	4 Octets	4 Octets	4 Octets	48 Octets
Source	Dest	Type	CRC	DATA

Ces paquets sont de taille fixe (***64 octets***) et possèdent cinq champs, soit une **source** indiquant la provenance du paquet, une **destination**, un **type** pour la qualité de service, un code **CRC** pour la détection d'erreur et finalement les **données** transportées. La définition de cette structure vous sera fournie.

3. Plateforme matérielle

La flexibilité des puces multiprocesseurs configurables Zynq utilisées en laboratoire nous permet de les utiliser pour faire du multitâche asynchrone : tel que montré à la ***figure 2***, les deux processeurs de la puce se partagent les ressources du système et exécutent chacun respectivement Linux et µC-OS. Ce genre de configuration dite AMP (Asymmetric Multi-Processing)[1] permet d'avoir un système d'exploitation plus général et offrant plus de fonctionnalités pour exécuter les tâches non critiques tout en gardant un processeur pouvant exécuter des tâches en temps réel pour les tâches devant

[1] Par opposition à SMP (Symmetric Multi-Processing) où un OS partagé contrôle tous les processeurs et permet l'exécution de n'importe quel tâche ou application sur n'importe lequel d'entre eux.

garantir un temps de réaction maximal. Nous verrons en classe plus en détails l'architecture de la Figure 2.

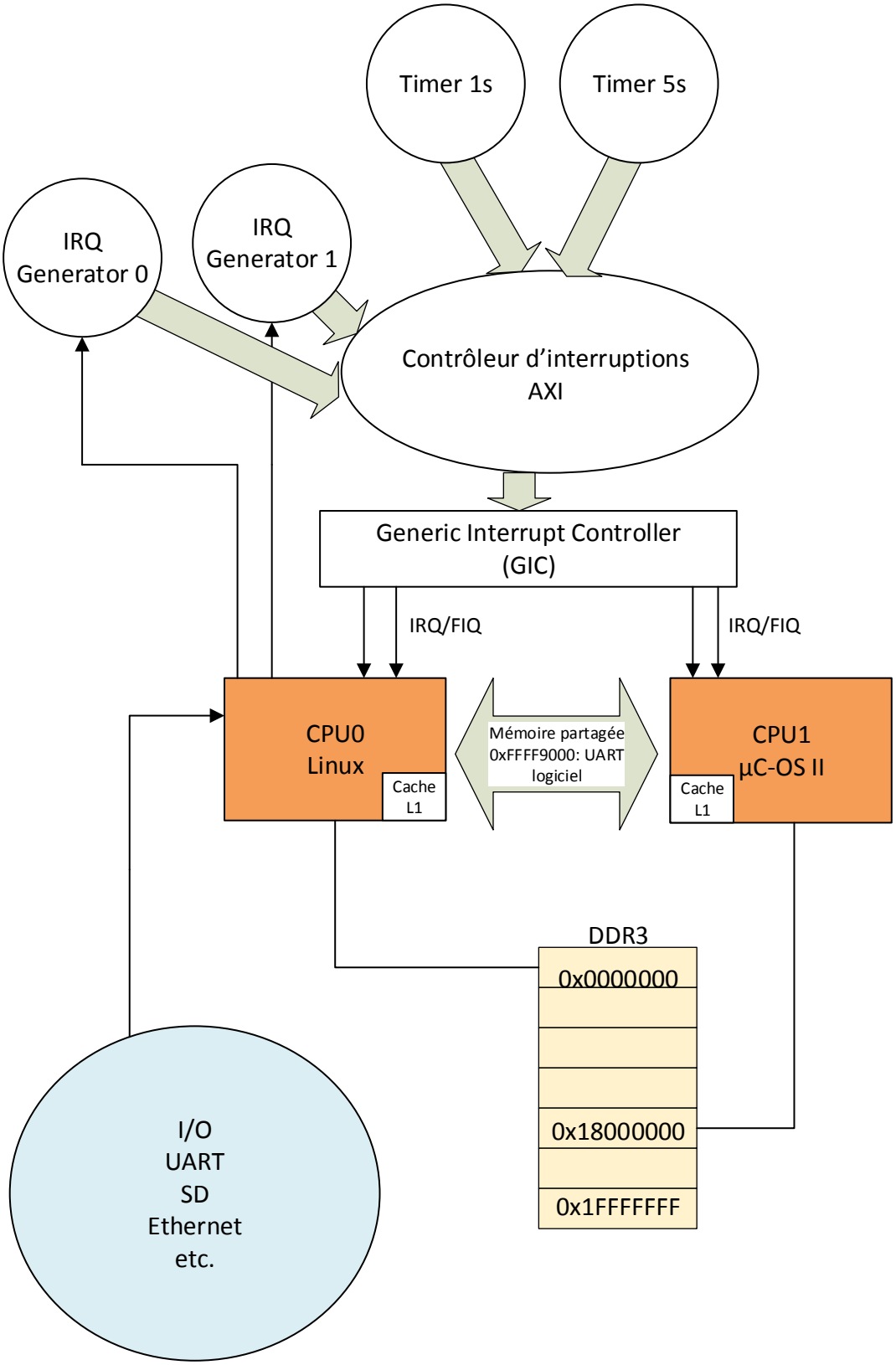


Figure 2 Configuration matérielle

4. Description des tâches

La ***figure 3*** illustre le flot des paquets à l'intérieur du routeur que vous devez implémenter sur le CPU µC-OS de la carte. La partie Linux, quand-à-elle, est fournie et sert à générer et à envoyer les paquets à traiter par le routeur.

Voici une brève description des différentes tâches :

Réception des paquets

La tâche est à concevoir. Elle attend sur un sémaphore donnée par la routine d'interruption de *irq_gen_0*, déclenchée par le processeur Linux pour signifier qu'un paquet est prêt. Le paquet de 64 bytes doit être lu long par long (4 bytes) à l'adresse *COMM_RX_DATA* et le processeur Linux doit être informé que chaque long a été consommé en lisant 0 à l'adresse *COMM_RX_FLAG* (ce flag sera remis à 1 lorsque le prochain byte sera disponible). Le débit d'arrivé des paquets est fixe à 1 paquet par seconde. Les adresses de destination des paquets sont aussi choisies aléatoirement, de même que le type de donnée de ces paquets. Les paquets créés doivent ensuite être placés dans une queue qui sera consommée par la tâche Calcul. Les flags sont définis dans *bsp_init.h*.

Calcul

La tâche est à concevoir. Elle doit répondre à plusieurs critères :

- Dans un premier temps, elle doit valider la provenance des paquets. Ainsi, tous les paquets provenant d'une plage d'adresse à rejeter doivent être rejetés .Ces plages vous sont fournies (defines commençant par `REJECT_`).
- Une fois l'adresse vérifiée, il faut s'assurer que le paquet n'a pas été corrompu. Il faut donc tester le CRC. La fonction qui calcule le CRC vous est fournie. Pour plus de renseignements sur le sujet veuillez lire : http://www.faqs.org/rfcs/rfc1071.html. La première utilisation de la fonction `checksum()` crée le checksum. Si l'on utilise à nouveau la fonction `checksum()` sur un paquet dont le checksum est défini, la fonction retourne 0 si le paquet n'est pas corrompu.
- Enfin, les paquets doivent être envoyés dans différentes files selon le type de paquets (vidéo, audio et autres resp. 1,2 et 3. Si les files sont pleines, les paquets excédants sont rajoutés à la file lue par la tâche **vérification.** Si jamais cette file est aussi pleine, vous devez détruire le paquet.

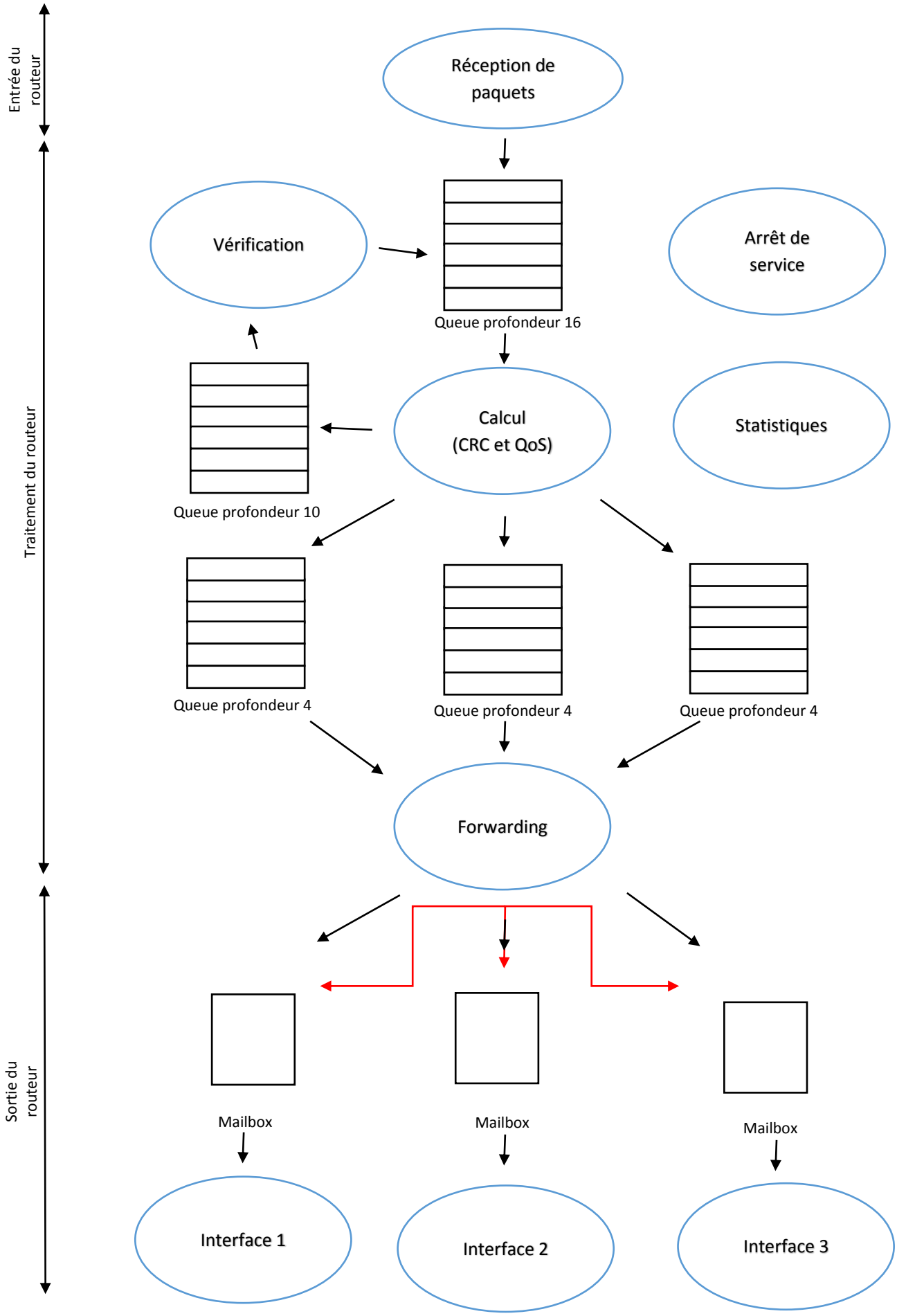


Figure 3 Flot de données dans le routeur

Forwarding

La tâche est à concevoir et doit répondre à ces critères :

- La tâche doit lire dans les trois queues en respectant la priorité définie par cette qualité de service : les paquets vidéo sont plus prioritaires que les paquets audio qui sont eux-mêmes plus prioritaires que les autres paquets.
- La tâche doit ensuite lire l'adresse de destination du paquet dans une table de routage. Dans les routeurs complexes, cette table est souvent réalisée en matériel. Ici, nous simplifions cette étape : le premier quart des adresses correspond à un *broadcast* sur les 3 interfaces, le second quart correspond à l'interface 3, le troisième quart correspond à l'interface 2 et le dernier à l'interface 1. Un délai de 2 secondes doit être ajouté afin de simuler l'accès à la table :

    - 0 < **Destination** < 1073741823 -> BROADCAST
    - 1073741824 < **Destination** < 2147483647 -> interface 3
    - 2147483648 < **Destination** < 3221225472 -> interface 2
    - 3221225473 < **Destination** < 4294967295 -> interface 1

    NB : Pour les paquets broadcastés, faites bien attention à allouer de l'espace pour les paquets nouvellement créés

Interface (PRINT)

La tâche est à concevoir. Cette tâche peut représenter le périphérique d'arrivée ou un autre routeur. Ici, elle se contentera de lire les paquets et de les imprimer.

Arrêt de service

La tâche est à concevoir. Cette tâche va se réveiller toute les secondes grâce à une synchronisation unilatérale avec l'interruption générée par le *fit_timer_1s*. Elle a pour charge de vérifier le nombre de paquets rejetés pour cause de mauvais CRC. Si le nombre est >= à 15, Alors elle arrête le routeur à l'aide de la fonction *OSTaskDel()*.

Vérification

La tâche est à concevoir. Cette tâche va se réveiller toute les 5 secondes grâce à une synchronisation unilatérale avec l'interruption générée par le *fit_timer_5s*. Elle a pour charge de remettre dans la file les paquets mis de côté pour cause de files trop pleines (vidéo, audio ou autres).

Statistiques

La tâche est à concevoir. Cette tâche est réveillée de manière asynchrone grâce à une synchronisation unilatérale avec une seconde interruption générée par Linux à l'exécution du script */mnt/printStats.sh*. Cette tâche va tout simplement afficher les informations de paquets traités au moment de son appel.

NB : Attention à protéger les *xil_printf()* avec des exclusions mutuelles car cette tâche peut se déclencher n'importe quand.

4. Travail à effectuer

4.1 Logiciel

Pour servir les interruptions générées par les minuteries et Linux, vous devrez implémenter des fonctions *handler*[2] qui seront appelées à chaque fois qu'une interruption sera générée. Vous avez à modifier les fichiers suivants :

- Bsp_init.h : déclaration de vos *handlers* :

```
void VotreInterruption(void* InstancePtr) ;
```

- Bsp_init.c : connexion de vos fonctions au contrôleur d'interruptions et activation de l'interruption dans le contrôleur d'interruptions
- Cpu1_uc.c : définition des fonctions *handlers* associées à vos interruptions et des tâches µC-OS.

Les *handlers* des 2 *fit_timer* et des 2 interruptions provenant de Linux génèrent des synchronisations unilatérales par sémaphore avec les tâches vérification, arrêt de service et impression des statistiques.

Enfin, vous devez implémenter les tâches décrites plus haut. La fonction de calcul de CRC vous est fournie, de même que le code Linux (qui ne doit pas être modifié). La configuration matérielle du Zynq a aussi déjà été générée. Vous n'avez donc qu'à utiliser la *workspace* Xilinx SDK fournie. La figure 3 **doit** être respectée lors de l'implémentation.

En plus de répondre au cahier des charges des tâches citées plus haut, votre code doit gérer de manière sommaire les cas d'erreurs ainsi que protéger les variables partagées si il y a besoin.

Voici un déroulement proposé :

1. Ne créez que les tâches liées aux interruptions et les *handlers* associés
2. Connectez vos *handlers* aux interruptions

[2] En classe on utilise le terme *myISR*, c'est la partie propre à l'ISR.

3. Testez vos interruptions (Est-ce que les interruptions fonctionnent ? Est-ce que les tâches associées démarrent ? Est-ce que les interruptions sont bien servies (acknowledge)?
4. Codez les tâches du flot principal (*receivePacket*, *computing*, *forwarding* et *print*)
5. Testez le flot principal avec seulement l'interruption de *receivePacket* connectée
6. Rajoutez une par une les interruptions et le code correspondant dans les tâches restantes (stats, stop, verif)

4.2 Affichage de trace pour débogage sous ARM

Il est interdit d'utiliser le fameux **printf()**. En effet, le laboratoire porte sur la programmation dans un système embarqué. La mémoire sur FPGA étant limitée, il faut une taille de code petite. La fonction **printf()** classique prend *55 Ko* en mémoire. À la place, vous utiliserez **xil_printf().** Cette fonction se comporte exactement comme **printf()** mais ne prend que **2 Ko** de mémoire.

5. Compléments

5.1 A propos des interruptions

Lors de ce laboratoire, vous avez à composer avec des interruptions matérielles. Quelques éclaircissements sur ces interruptions semblent nécessaires.

5.1.1 Le contrôleur d'interruptions

Pour ce laboratoire nous n'utilisons pas directement le contrôleur d'interruption déjà présent sur la carte (le GIC). Le détail de cette implémentation est présent dans le code mais vous n'avez pas à vous en soucier. Le contrôleur que nous utilisons est un contrôleur matériel, *AXI_intc* qui est lui-même connecté à un IRQ du GIC laissé libre à cette fin. Autrement dit, le GIC laisse passer l'interruption (Figure 2), mais c'est *AXI_intc* qui va la gérer. C'est donc à ce contrôleur que vous serez confrontés lorsque vous devrez connecter vos *handlers*, mais aussi lorsque vous devrez signaler que vos interruptions ont bien été servies. Vous pouvez trouver l'intégralité des fonctionnalités du contrôleur d'interruption matériel dans les fichiers inclus par le bsp dans SDK (app_cpu1_bsp/libsrc/intc_v2_05_a/src/xintc.c).

5.1.2 Les interruptions générées par Linux

Deux de nos interruptions sont déclenchées lorsque Linux écrit dans certaines adresses en mémoire (0x78600000 et 0x78620000). Ces adresses correspondent en réalité aux adresses de bases de deux modules matériels (irq_gen_0 et irq_gen_1). Ces modules sont en fait deux registres 32 bits. Le bit situé en première position (les adresses citées plus haut) est le seul utilisé et définit si le port interruption du module est mis à HIGH. Ainsi, vous devrez, en plus de signaler au contrôleur

d'interruption que l'interruption a bien été servie, signaler au module matériel que l'interruption est terminée. Pour ce faire, vous devrez nettoyer le registre du module à l'aide de la fonction `Xil_Out32()` qui vous permet d'écrire un long à une adresse donnée de l'espace d'adressage. A noter que les données d'adressage sont contenues dans les structures `fpga_core` données dans le code (`m_irq_gen_0` et `m_irq_gen_1`) que vous pouvez passer en paramètre à vos *handlers*.

5.1.3 Les interruptions générées par les minuteries (timers)

Ces interruptions sont moins compliquées à servir car ils ne font que générer une interruption à intervalle fixe. Ainsi, côté matériel, les interruptions n'ont pas besoin d'être remise à 0. Le contrôleur d'interruption doit tout de même être prévenu dès que l'interruption est servie.

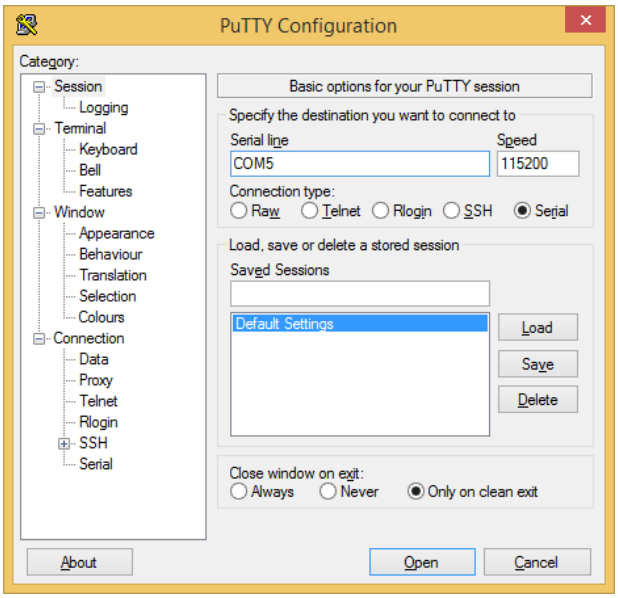
5.2 Ouverture du projet

Le code contient un *workspace* Xilinx SDK (eclipse). Vous devez donc simplement pointer vers ce dossier à l'ouverture du SDK.

Note : Pour sauver de l'espace, les fichiers compilés de la *workspace* ont été supprimés avant de vous les remettre. Pour chaque projet, vous devez donc faire un clic droit sur le projet, sélectionner *Refresh* puis recliquer et sélectionner *Clean Project* pour forcer leur régénération.

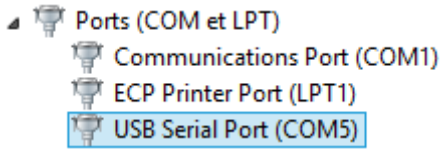
5.3 Envoi du code sur la carte

Le contenu du dossier SDCARD doit être copié à la racine de la carte SD. Le contenu de ce dossier est automatiquement mis à jour à la compilation du projet SDK. Par la suite, un terminal vers la partie Linux doit être ouvert, ce qui peut être fait à l'aide de Putty, qui peut être téléchargé ici. Il suffit ensuite d'ouvrir ce programme, de choisir une connexion de type *Serial*, sur COM5* à 115200 bauds, puis de cliquer sur *Open*.



Par la suite, il suffit d'exécuter le script */mnt/start.sh* pour commencer à envoyer des paquets. Notez que la sortie de la fonction *xil_printf()* sera directement affichée sur le terminal Linux.

* Windows et déterminisme ne rimant pas ensemble, il est fort possible que la planchette de développement soit sur un autre port que COM5. Pour savoir si c'est le cas, ouvrez le *Gestionnaire de périphériques* de Windows et cherchez le port du *USB Serial Port*.



5.4 Débogage

L'utilisation de la carte SD n'est nécessaire que pour démarrer le système. Par la suite, le débogage et l'envoi d'une nouvelle version de l'application µC-OS est possible en suivant ces étapes :

1. Ouvrir xmd, soit avec Xilinx Tools -> XMD Console ou avec Xilinx Tools->Launch Shell -> xmd
2. Dans xmd, lancer la commande *connect arm hw –debugdevice cpunr 2*
3. Déboguer avec éclipe en cliquant sur la flèche à droite de l'icône de debug ->app_cpu1 Debug

6. Questions

6.1 Expliquez brièvement la logique qui vous a conduit à choisir les priorités de vos tâches.

6.2 Quelle est l'utilité du fichier source *bsp.c* et pourquoi en a-t-on besoin ici?

6.3 Il vous est demandé dans le rapport d'implémenter 4 fonctions *handler*. Ces fonctions ont pour but de réveiller des tâches par synchronisation unilatérale. Aurait-on pu mettre le contenu des tâches directement dans les *handlers* ? Quels auraient été les avantages/inconvénients d'une telle méthode?

6.4 Avec les données qui vous ont été fournies dans le rapport, spécifiez le débit théorique maximal de paquets pouvant transiter dans votre routeur. Est-ce que le débit est suffisant pour assurer le traitement des paquets envoyés par la tâche d'injection de manière continue ? Comment pourrait-on augmenter le débit de notre système sans modifier les valeurs numériques de l'énoncé ?

6.5 La configuration matérielle actuelle du laboratoire connecte les périphériques (UART, GPIO, Ethernet, etc.) sur le processeur Linux plutôt que sur le processeur µC-OS. D'après-vous, est-ce une bonne idée? Serait-t-il plutôt préférable connecter tous ces périphériques sur le processeur temps réel?

7. Rapport

Vous avez à écrire un rapport contenant les réponses aux questions ainsi que les points de votre implémentation qui vous semblent importants. Concernant les explications sur votre implémentation, soyez brefs, il s'agit pour nous de comprendre votre démarche si certains points vous paraissent obscurs. Vous terminerez votre rapport par une brève critique du laboratoire ainsi que le nombre d'heures passées sur celui-ci.

8. Procédure de remise

Vous devez remettre à la fin de ce laboratoire un fichier .zip ou .rar contenant un répertoire avec le code et un fichier .doc/.docx/.odt avec le rapport. Veuillez mettre dans le nom du rapport et de l'archive le texte suivant : « Lab2_A15_matricule1_matricule2 ». N'envoyez que les fichiers source que vous avez modifiés.

Date de remise limite : 22 octobre pour le groupe B1, 29 octobre pour B2.

9. Pénalités

Pénalités	
Retard	-1.5 points par jour ouvrable, note 0 si 4 jours de retard
Fichiers mal présentés ou orthographiés	Jusqu'à -1 point
Source non compilable	Assurez-vous que votre projet compile, sinon vous obtiendrez la note 0 pour l'exécution
Remise	Fichiers mal nommés = -1 point

Références

1. XAPP1078 : *Simple AMP Running Linux and Bare-Metal System on Both Zynq SoC Processors*