Laboratoire 2 – INF3610

SIMULATEUR D'UNE TOUR DE CONTRÔLE D'AÉROPORT

Objectifs

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 et du temps réel
- S'initier aux environnements de développement embarqués, tels Xilinx SDK
- Étudier les spécificités de la programmation pour SoC.
- Utiliser et comprendre un mécanisme d'interface entre modules logiciels et matériels.

Plateforme matérielle

La flexibilité des puces multiprocesseurs configurables Zynq utilisées en laboratoire nous permet de les utiliser pour faire du multitâche synchrone ou asynchrone : tel que montré à la Figure 1: Configuration matérielleFigure 1, les deux processeurs de la puce se partagent les ressources du système et pourraient exécuter des systèmes d'exploitation différents, tels que Linux ou µC-OS (sur chacun un cœur différent), ou Linux sur les deux cœurs, par exemple.

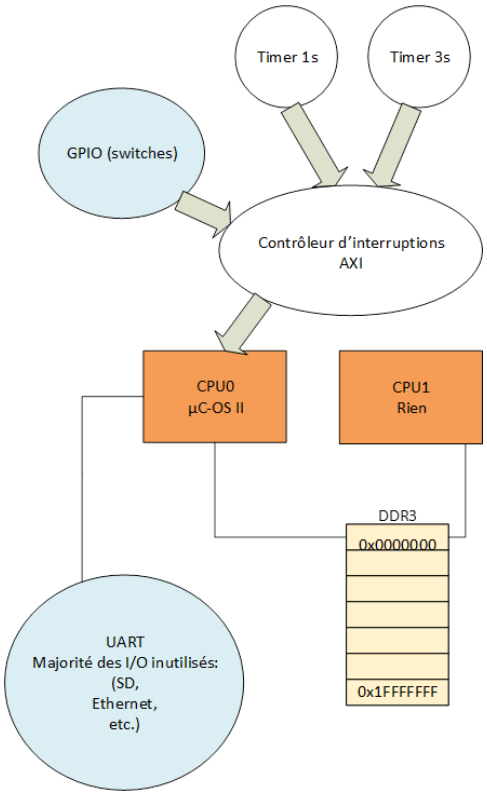

Figure 1: Configuration matérielle

Le premier genre de configuration dit AMP (Asymmetric Multi-Processing) 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 garantir un temps de réaction maximal. Par contre, dans le deuxième genre de configuration, dit SMP (Symmetric Multi-Processing), 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, facilitant la programmation du système au détriment de la flexibilité offerte par l'AMP. Nous verrons en classe plus en détails l'architecture de la Figure 2 et ces deux types de configuration. Dans ce laboratoire, nous n'utiliserons qu'un seul processeur exécutant µC-OS.

Mise en contexte

Ce laboratoire se veut une simplification d'un simulateur d'une tour de contrôle dans un aéroport qui a pour but d'assurer la synchronisation de plusieurs avions. La figure suivante illustre les différents concepts mis en application dans ce laboratoire :

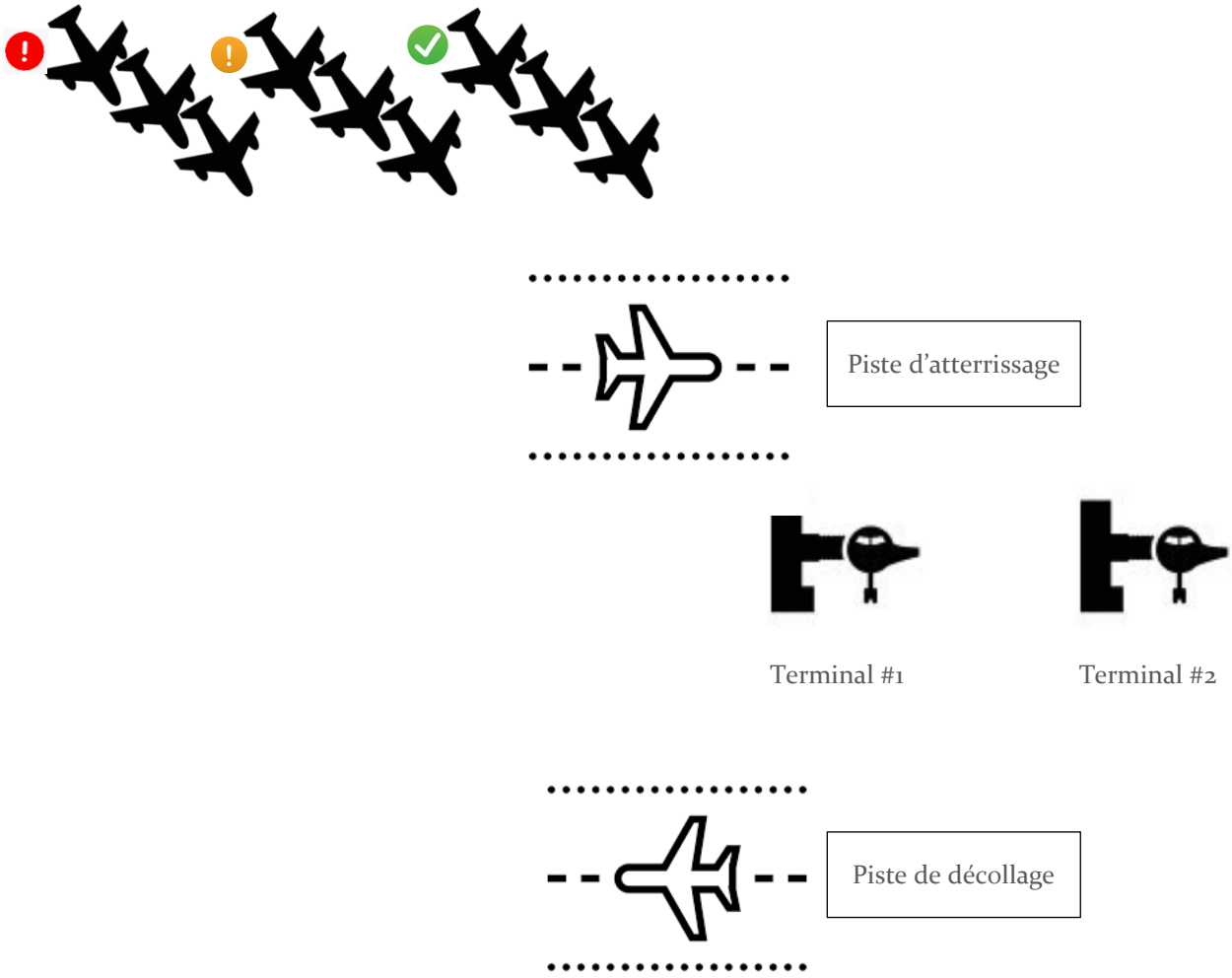

Figure 2 : Schéma processus

EXPLICATION DU FLOT

Le laboratoire simulera les mouvements des avions dans un (très petit) aéroport selon le flot suivant :

- Les avions en vol qui sont à proximité de l'aéroport doivent attendre d'obtenir la permission d'atterrir sur la piste d'atterrissage. Cet aéroport ne contient qu'une seule piste d'atterrissage.

    - Selon le temps de retard qu'ils ont accumulé, certains avions seront plus prioritaires que d'autres. 3 priorités seront définies :
        - Urgent (HIGH) : avions ayant entre 40 et 60 min de retard (nous assumons qu'il n'y a pas d'avions ayant plus de 1h de retard)
        - Pressant (MEDIUM) : avions ayant entre 20 et 39 min de retard
        - À l'heure (LOW) : avions ayant moins de 20 min de retard
- Une fois la permission donnée à un avion d'utiliser la piste d'atterrissage, ce dernier devra atterrir (et oui!), et attendra qu'un terminal se libère.
- 2 terminaux sont disponibles afin de faire descendre les passagers de l'avion, nettoyer l'avion, et faire monter les passagers du vol suivant. Si aucun terminal n'est disponible, l'avion doit attendre sur la piste d'atterrissage (aucun autre avion ne peut atterrir pendant ce moment).
- Une fois les nouveaux passagers à bord, l'avion attendra que la piste de décollage soit libre à un endroit dédié à cette attente (les terminaux et la piste de décollage peuvent être utilisés par d'autres avions pendant ce temps). Ensuite, l'avion pourra utiliser la piste de décollage.

Travail à effectuer

ÉTAPE 1 : IMPLÉMENTATION MATÉRIELLE

Suivez le tutoriel *Création d'un projet Vivado.pdf* joint avec cet énoncé pour la création de la plateforme matérielle avec Vivado, et la création du BSP (*Board Support Package*) sur Xilinx SDK.

ÉTAPE 2 : IMPLÉMENTATION LOGICIELLE

Poursuivre l'implémentation avec la partie logicielle. La procédure suivante est proposée :

1. Interruptions

Pour servir les interruptions générées par les minuteries (*fit_timer* de 1 sec et 3 sec) et les switches, vous devrez implémenter des fonctions *handler*[1] qui seront appelées à chaque fois qu'une interruption sera générée. Vous avez à modifier les fichiers suivants :

---

[1] En classe on utilise le terme *myISR*, c'est la partie propre à l'ISR. En d'autres termes, les *handlers* d'interruption peuvent être vus comme des *callbacks* exécutés lors d'une interruption.

- bsp_init.h :
    - Ajout de la déclaration de vos fonctions de connexion et de déconnexion d'interruption.

Remarquez également la déclaration des *handlers* des interruptions ayant la signature suivante:

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

- bsp_init.c :
    - Implémentation de vos fonctions de connexion et déconnexion d'interruption ;
    - Appel des fonctions de connexion et déconnexion aux endroits pertinents ;
    - Implémentation de la fonction initialize_gpio(), en faisant les appels nécessaires pour activer l'interruption générée par les switches. Voir la section *Compléments* à la fin de ced document pour plus d'informations sur le driver GPIO.
- simAvion.c :
    - Définition des fonctions *handlers* associées à vos interruptions des *fit_timer* et des switches. Ces interruptions génèrent des synchronisations unilatérales par sémaphore avec les tâches génération de données, vérification et statistiques.

Astuce : Une fois les modifications ci-dessous faites, assurez-vous que les interruptions fonctionnent correctement. Pour cela, simplement faire un xil_printf dans les *handlers* du fichier simAvion.c (que vous enlèverez par la suite). Cela vous permettra aussi de vous assurer que la sortie de votre programme est bien configurée…

2. Tâches du simulateur

Ensuite, il faudra implémenter les fonctions du simulateur. Ces fonctions sont décrites en détails dans la section suivante. De plus, veuillez prendre en considération les quelques notes suivantes lors de l'implémentation de votre programme.

Note sur les xil_printf

Il est interdit d'utiliser le fameux **printf()**. En effet, le laboratoire porte sur la programmation d'un système embarqué. La mémoire sur FPGA étant limitée, il faut minimiser la taille du code. 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.

Notes sur les mutex

- Attention à protéger les appel à *xil_printf()* avec des exclusions mutuelles car cette tâche peut se déclencher n'importe quand.
- Idem pour les appels aux fonctions malloc et/ou free, car l'implémentation de ces fonctions suppose une configuration *bare metal* et n'est donc pas *thread-safe*.

Attention! Ces précautions doivent être appliquées même pour le code qui est fourni.

Suggestion : Vous pouvez vous créer des fonction safe_print et safe_malloc/free qui s'occuperont d'acquérir et libérer le mutex au lieu de le faire à maintes reprises dans votre programme.

- D'autres mutex peuvent être nécessaires pour la protection de diverses variables. À vous d'en juger de la pertinence.

Notes sur les priorités

Vous ne pouvez pas modifier la priorité des tâches. Cependant, si vous allouez des mutex, vous devrez définir leur priorité en conséquence.

Notes sur la gestion des erreurs

Votre code doit gérer de manière sommaire les cas d'erreurs (comme au laboratoire 1).

ÉTAPE 3 : RAPPORT

Un rapport PDF contenant les réponses aux questions théoriques présentés à la fin de ce document est demandé. Vous pouvez également détailler les points de votre implémentation qui vous semblent importants, si nécessaire (non obligatoire). Soyez brefs, il s'agit simplement pour nous de comprendre votre démarche si certains points vous paraissent obscurs. Finalement, il serait apprécié de nous fournir une brève critique du laboratoire ainsi que le nombre d'heures passés sur ce dernier afin d'améliorer le laboratoire pour les sessions à venir.

Description des tâches

Le squelette de chaque fonction vous est fourni, et des indications pour les compléter sont fournies sous la forme de « TODO ».

GÉNÉRATION DE DONNÉES

Cette tâche est réveillée de manière asynchrone grâce à une synchronisation unilatérale avec une interruption générée par le timer de 1 seconde (fit_timer_1s). Cette tâche s'occupe de générer des « avions » avec des valeurs aléatoires pour la destination, l'origine, et le temps de retard. Une fois les valeurs générées, la tâche mettra les avions dans les files appropriées selon leur temps de retard (priorité HIGH, MEDIUM ou LOW).

Afin de laisser le temps aux avions de passer à travers le processus et afin de ne pas inonder le système, la tâche génération décidera, de façon aléatoire, de « sauter » une génération en moyenne 1 fois sur 5 (implémentation fournie).

ATTERRISSAGE

Cette tâche doit se mettre en attente des 3 files contenant les avions prêts à atterrir et donner la permission à l'avion ayant la priorité la plus élevée.

Une fois l'avion « choisi », la tâche devra attendre que l'avion atterrisse (OSTimeDly), et se mettra en attente d'un terminal libre. Vous utiliserez le mécanisme de synchronisation que vous jugerez pertinent entre les terminaux et l'atterrissage.

L'avion sera ensuite transféré au terminal libre grâce au mécanisme de votre choix.

TERMINAL

Chaque terminal attend de recevoir des avions selon le mécanisme choisi. Une fois l'avion reçu, il faudra attendre que ce dernier se vide de ses passagers (OSTimeDly) et il faudra regénérer les données pour le nouveau vol.

Ensuite, il faudra mettre l'avion dans la file afin de pouvoir accéder à la piste de décollage.

Cet aéroport comporte 2 terminaux. Il faudra donc s'assurer de créer 2 tâches effectuant cette fonction, sans toutefois écrire 2 fonctions différentes.

DÉCOLLAGE

La tâche de la piste de décollage devra attendre un avion à faire décoller. Il faudra ensuite prendre le temps de faire décoller l'avion (OSTimeDly). Finalement, il faudra détruire l'avion.

STATISTIQUES

Cette tâche est réveillée de manière asynchrone grâce à une synchronisation unilatérale avec l'interruption générée par la modification de l'état de n'importe laquelle des switches du Zedboard. Cette tâche doit obtenir des statistiques par rapport à l'état de l'aéroport au moment où la tâche est appelée. Concrètement, il est question d'afficher le nombre d'avions en attente d'atterrissage en fonction de leur priorité, le nombre d'avions en attente de décollage et l'état des terminaux (libre ou occupé).

VÉRIFICATION

Cette tâche est réveillée de manière asynchrone grâce à une synchronisation unilatérale avec une interruption générée par le timer de 3 secondes (fit_timer_3s). Elle vérifie qu'il n'y a pas d'avions « refusés » soit des files d'attente d'atterrissage qui débordent. Concrètement, cela implique que si la tâche de génération tente d'envoyer des avions sur une des trois files d'atterrissage, mais que cette file est pleine, elle devra mettre le booléen stopSimDebordement à true. La tâche vérification se chargera de vérifier la valeur de cette variable et de suspendre (OSTaskSuspend) **toutes** les tâches.

Questions théoriques

QUESTION 1

Lors de l'implémentation matérielle, vous avez instancié un module GPIO, 2 FIT Timers et un contrôleur AXI. Donner une brève définition et une explication du rôle de chaque module dans ce laboratoire.

QUESTION 2

Combien de périphériques (minuteries, switches, etc.) aux maximum votre design pourrait-il supporter?

QUESTION 3

Au cours no 4 (pages 23 à 25), on a présenté le déroulement générique des interruptions uC, alors qu'au cours no 5 (pp. 25 à 34) on a présenté la gestion des interruptions propres au Zynq. À partir de cette information et des fichiers du portage du lab 2 (e.g. bsp_init.c et os_cpu_a.S) ainsi que que votre

code d'application (simAvion.c) décrivez de manière la plus précise possible le déroulement complet (différentes étapes) d'une interruption provenant de fit timer o jusqu'à son service ISR.

Remise

Ce laboratoire est à remettre 4 semaines après la première période de laboratoire (voir Moodle pour la date de chaque groupe).

Vous devez remettre fichier .zip contenant un répertoire avec le code et un fichier PDF avec le rapport. Veuillez mettre dans le nom du rapport et de l'archive le texte suivant : « Lab2_A18_matricule1_matricule2 ». Les fichiers source à remettre sont les fichiers simAvion.c/h et bsp_init.c/h. **La plateforme matérielle que vous avez créée n'est pas nécessaire.**

Barème	
Exécution du code	
Fonctionnalités	/10
Qualité du code	/4
Réponse aux questions théoriques	
Question 1	/2
Question 2	/2
Question 3	/2
Le non-respect des consignes entraine des points négatifs (peut aussi invalider les points d'un exercice)	
TOTAL	/20

Compléments sur les interruptions

CONTRÔLEUR D'INTERRUPTIONS

Pour ce laboratoire nous n'utilisons pas directement le contrôleur d'interruption déjà présent sur la carte (GIC pour *Generic Interrupt Controller*, qui est présent sur une grande portion des implémentations de processeur ARM). 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 (*Coreo_nIRQ* dans Vivado). **Autrement dit, c'est le GIC qui interrompt le processeur, mais c'est le *AXI_intc* qui va gérer**

l'interruption. C'est donc à ce contrôleur que vous serez confrontés lorsque vous devrez connecter vos *handlers* et activer vos interruptions, mais aussi lorsque vous devrez signaler que vos interruptions ont bien été servies. Ceci permet de multiplexer les 3 interruptions provenant du FPGA en un seul signal d'interruption du GIC.

Vous pouvez donc vous baser sur le code connectant l'interruption GIC du AXI_intc (fonction connect_intc_irq) pour connecter vos interruptions au AXI_intc, mais assurez-vous de garder en tête que vous devrez utiliser l'API du AXI_intc (définit dans le fichier *xintc.h* et ayant la forme XIntc_Connect) plutôt que celui du GIC (ayant la forme XScuGic_Connect).

Documentation du BSP

Le BSP de Xilinx, qui fournit les drivers XGpio[2], XIntc, etc. est documenté dans le code de celui-ci, dont les headers sont disponibles dans le projet *standalone_bsp_0/ps_cortexa9_1/include/{xgpio, xintc, etc.}.h*. Ceux-ci donnent un aperçu général des fonctions de cette partie spécifique du BSP, et une documentation plus spécifique est disponible dans l'implémentation, accessible par la sélection de la fonction dans son header et du raccourci F3.

INTERRUPTION GÉNÉRÉE PAR LE GPIO (SWITCHES)

Le driver XGpio de Xilinx doit être utilisée afin d'initialiser le GPIO et activer les interruptions. Notez qu'il vous faudra peut-être configurer le driver pour qu'il génère des interruptions lorsque l'état des switches est modifié. De même, dans votre *handler* d'interruption, il faudra l'**aviser que l'interruption a été servie**.

INTERRUPTION GÉNÉRÉES PAR LA MINUTERIE (TIMER)

Ces interruptions sont moins compliquées à servir car elles ne font que générer une interruption à intervalle fixe. Aucune modification logicielle n'a donc besoin d'être effectuée (outre l'activation de l'interruption et la connexion de son *handler*).

---

[2] Faites bien attention de ne pas confondre le driver XGpio, qui gère les broches GPIO provenant de la partie FPGA de la puce (ce qui est le cas des switches sur le Zedboard) et le driver XGpio_PS, qui gère les broches GPIO contrôlés par la partie processeur ARM de la puce (non utilisés ici).