Polytechnique de Montréal - DGIGL

Laboratoire #1 : Routeur sur puce FPGA

INF3610 – Automne 2024

Séance 3 : Partitionnement de l'application de routage sur 2 cœurs en mode AMP

1. Objectif de la séance no 3

Une architecture multiprocesseur où tous les processeurs ne sont pas traités de la même façon par le système d'exploitation est dite asymétrique (AMP pour Asymmetric MultiProcessing). Par exemple, un système peut autoriser (que ce soit au niveau du matériel ou du système d'exploitation) à un seul processeur d'exécuter le code du système d'exploitation ou peut autoriser un seul processeur à effectuer des opérations d'E/S. Un autre exemple de système AMP est un système multiprocesseur où chaque processeur roule son propre OS ou RTOS plutôt qu'un seul OS qui gère tous les processeurs[1]. Dans le cas de la séance no 3 on aura 2 ports (BSP) de uC/OS-III qui roulent chacun sur un core (core 0 et core 1)[2].

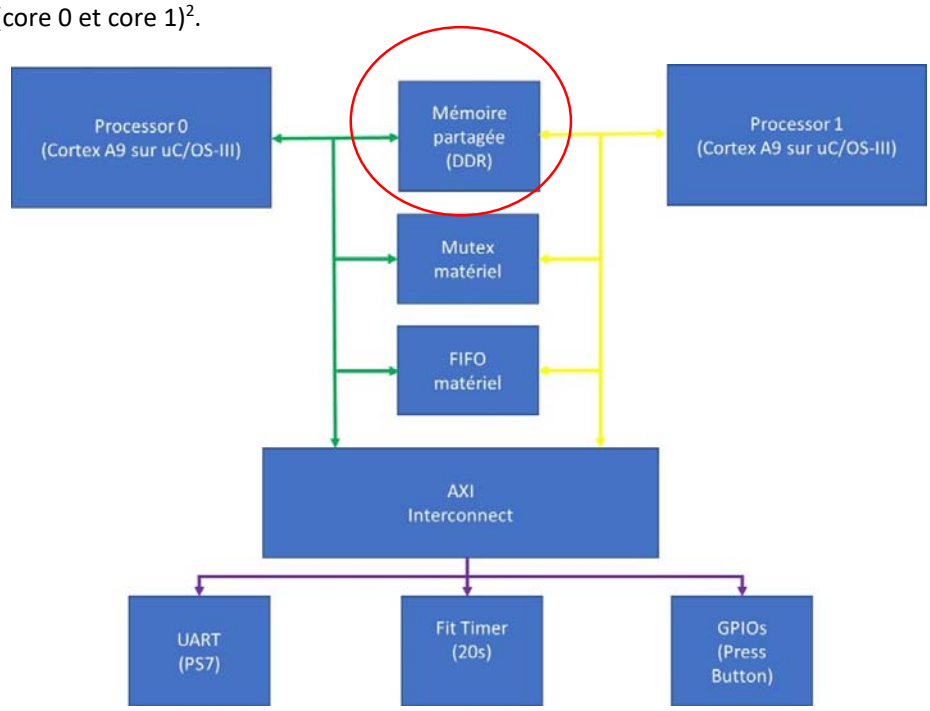


Figure 1. Architecture ciblée

Comme avec 2 processeurs on a 2 mémoires distinctes, la difficulté des systèmes AMP (mais aussi vrai pour SMP) est l'échange d'information entre les cores (ici core 0 et core 1). La figure 1 illustre 3 mécanismes pour faire cette communication sur PYNQ-Z2: 1) par mémoire partagée DDR, 2) par mutex matériel et 3) par FIFO matériel. Dans ce laboratoire, nous nous concentrerons sur une mémoire partagée dans la DDR (cercle en rouge sur la figure 1).

[1] On parle alors d'un système SMP (Symmetric MultiProcessing) tel que Linux sur un i7.
[2] Notez qu'on aurait pu aussi avoir comme système AMP core 0 qui roule baremetal et core 1 qui roule uC/OS-III ou encore core 0 qui roule FreeRTOS et core 1 qui roule uC/OS-III. Un système AMP peut donc rouler des OS différents.

L'objectif de cette 3e séance se concentre principalement sur 2 manipulations :

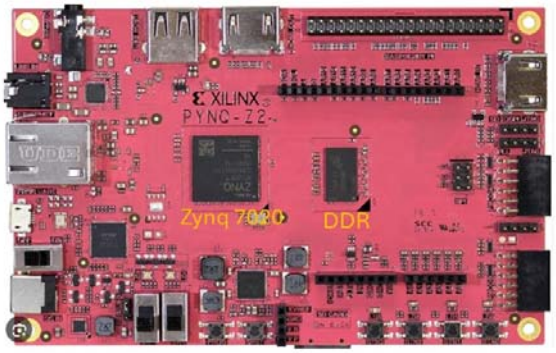
Réalisation 1) Passage de la tâche *TaskGenerate* du core 0 au core 1 et programmation AMP en fonction de ce nouveau partitionnement (c'est-à-dire core 0 avec tout les tâches et ISRs de la séance no 2 moins la tâche *TaskGenerate* et core 1 avec *TaskGenerate*) à l'aide d'une mémoire partagée.

Réalisation 2) Analyse de performance 2 cœurs vs 1 coeur

Réalisation 3) Utilisation d'une minuterie de type *watchdog* pour l'arrêt du système complet

Réalisation 4) Terminaison propre

Rappel important: la DDR se situe à l'extérieur de la puce Zynq 7020 :



2. Ce que vous devez savoir pour la réalisation 1 (incluant un tutoriel important pour la compréhension)

Concernant les adresses de programmes ELF sur core 0 et core 1

La figure 2 illustre l'assignation (mapping) des adresses sur une Zynq SoC série 7000. La PYNQ utilise seulement 512 MB puisque la DDR est limitée à cette valeur. D'autres cartes comme la ZYBO possèdent 1 GB.

Table 4-1: **System-Level Address Map**

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters[1]	Notes
0000_0000 to 0003_FFFF[2]	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU[3]
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFF_FFFF[4]	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF[2]	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

Figure 2 Mapping des adresses sur une Zynq SoC série 7000

Maintenant, les figures 3 et 4 illustrent les zones mémoires qu'utilise chaque core (core 0 et core 1). Sur la figure 3, on remarque que l'adresse de base (Base Address) choisie de la DDR pour le core 0 est 0x1000000 et que sa longueur est 0x1000000 (16 MB). Qu'est-ce que SDK va mettre (charger) dans ce 16 MB? La réponse est plus bas sur la figure 3, il s'agit du fameux fichier ELF[3] qui est en fait le résultat de la compilation. Notez que si on avait voulu loger une partie du programme ELF dans la RAM (p.e. RAM_0) on aurait pu le faire en remplaçant ps7_ddr_0 par ps7_ram_0.

[3] ELF pour *Executable and Linking Format* est un format de fichier binaire populaire (Linux) utilisé pour l'enregistrement de code compilé (objets, exécutables, bibliothèques de fonctions).

Du côté du core 1, on aura le linker script de la figure 4. Que remarquez-vous comme seul changement? Et oui, l'adresse de base (Base Address) fait suite à Base Adress de core 0 plus le size de core 0, ce qui donne l'adresse 0x2000000. Ceci nous assure que les 2 programmes peuvent s'exécuter en parallèle[4].

[4] SDK donne par défaut une valeur peu importe le core. C'est au concepteur de s'assurer qu'il n'y a pas de recouvrement entre 2 programmes ELF. Un recouvrement risque fort de faire planter l'exécution.

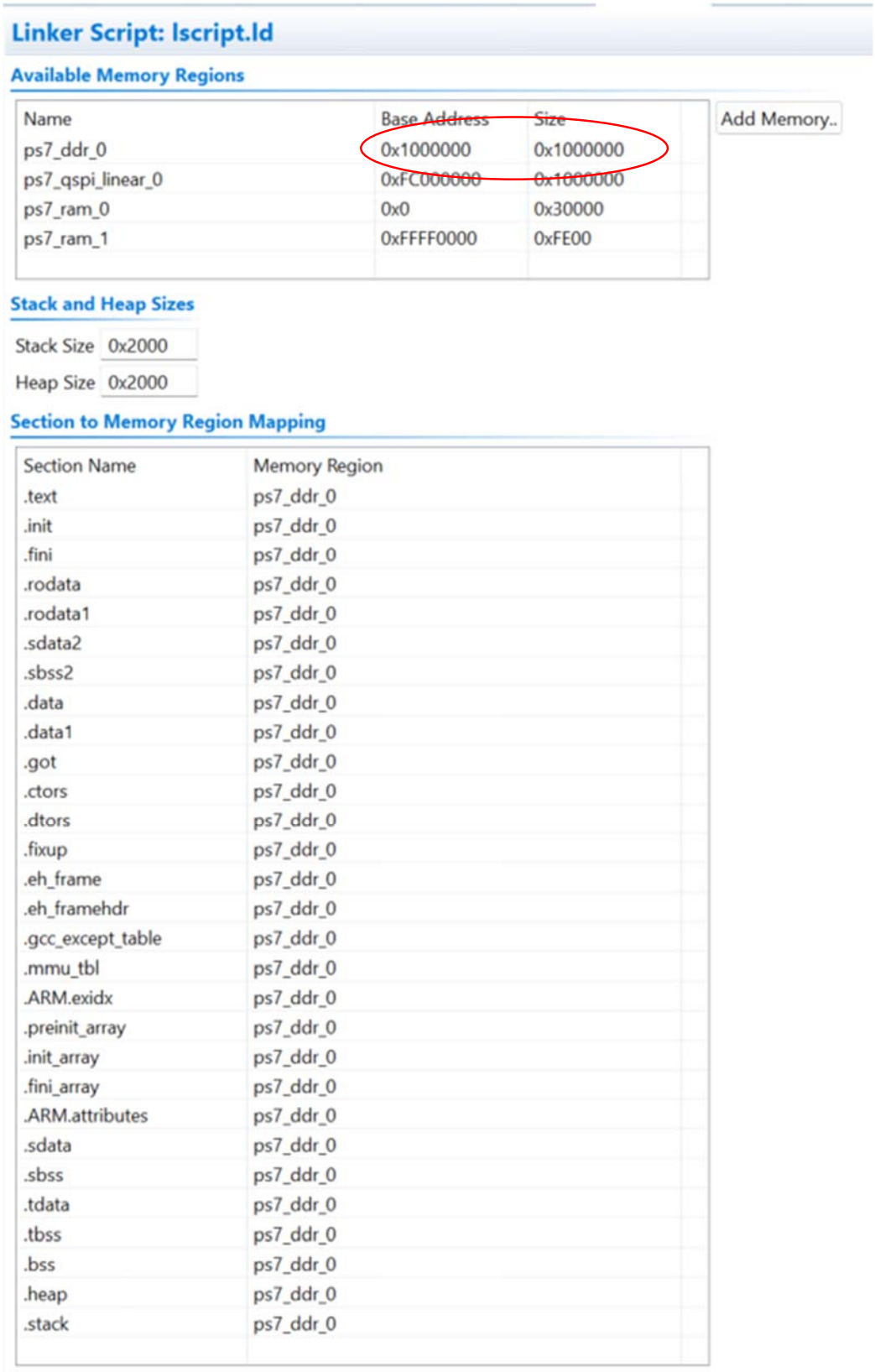


Figure 3 Linker script du core 0 qui contiendra tout le système de la séance no 2 moins *TaskGenerate*

Figure 4 Linker script du core 1 qui contiendra *TaskGenerate*

Comment créer la mémoire partagée DDR

On doit déterminer la plage d'adresse de la mémoire partagée (cercle rouge sur la figure 1). Celle-ci se fera forcément en dehors de celle des programmes de core0 et core 1. On pourrait par exemple décider de prendre une zone partagée à partir de l'adresse 0x3000000. Ce partage doit être annoncé dans un *define* sur chaque core :

```
const uint32_t BASEADDR = 0x3000000;
```

Comment vérifier que 0x3000000 est une bonne valeur? Deux choses : 1) on observe à la figure 2 que 0x3000000 est bel et bien réservée à la DDR et 2) les programmes ELF des figures 3 et 4 occupent de 0x1000000 à 0x2FFFFFF n'utilisent pas la plage 0x3000000 et plus (encore une fois pour éviter tout recouvrement). Nous allons donc créer une zone mémoire partagée entre core0 et core1 pour

échanger une rafale d'au maximum 255 paquets, donc 4K mots au maximum. À 4K mots on est inférieur à 3FFF FFFF, donc on est OK.

Dernier point, vous devez désactiver l'utilisation de la mémoire cache pour la zone qui sera partagée. Dans notre cas, pour des raisons de simplification nous désactiverons toute la cache juste avant OSStart() avec la commande *Xil_DCacheDisable().* **Important chaque programme ELF qui accède à la mémoire partagée doit faire cette désactivation.** Désactiver la cache entièrement est par contre couteux, on verra plus loin comment atténuer cet effet.

Comment synchroniser core 0 et core 1 lors d'un partage de rafale– tutorial du producteur/consommateur

Nous allons utiliser le protocole de type *handshacking* (poignée de main). Cela nécessite 2 signaux de contrôle *ack* et *req*. La figure 5 illustre le protocole entre un consommateur sur core 0 et un producteur sur core 1.

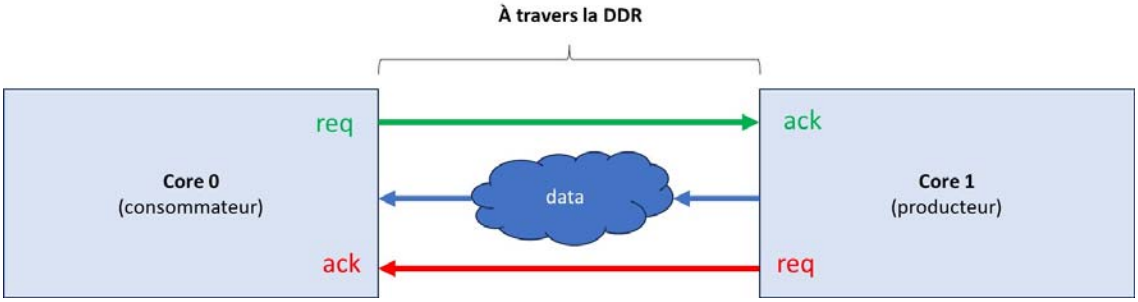


Figure 5 *Handshacking (échange avec poignée de main)* entre un consommateur et un producteur à travers la DDR : req de *core 0* et ack de *core1* sont la même adresse en DDR et même chose pour req et ack.

On suppose au départ que les variables *req*, *ack* définis en zone partagée dans la DDR sont à 0 et que la donnée (data) partagée est aussi dans la DDR. Un cycle de transfert se fera de la façon suivante :

i. *core 1* va produire une donnée et va attendre de manière active, avec *while (!ack)*, que *core 0* soit prêt à la consommer.
ii. Quand *core 0* est prêt à consommer la donnée, il met son *req* à 1 (ce qui débloque *core 1*) et il attend à son tour de manière active, avec *while (!ack)*, que core 1 lui donne le feu vert pour consommer.
iii. *core 1* met son *req* à 1 (ce qui débloque *core 0*) et va attendre de manière active que *core 0* est fini de consommer avec *while (ack)*.
iv. *core* 0 va consommer et puis il remet son *req* à 0 (ce qui débloquer core 1) pour indiquer qu'il a consommé et qu'on peut passer à une autre valeur.
v. core 0 attend de manière active, avec *while (ack)*, que core 1 lui dise de passer à une prochaine valeur à consommer.
vi. *core 1* met son *req* à 0 (ce qui débloque *core 0*) pour indiquer que c'est terminer et qu'il va passer à une prochaine donnée.
vii. Et on retourne à l'étape i pour une prochaine donnée.

Les programmes **producteurs** et **consommateurs** distribués comme code de départ illustrent un exemple de ce *handshacking*. Le transfert de *TaskGenerate* sera calqué sur cet exemple de producteur/consommateur mais au lieu de transférer des entiers (valeur i), vous transférez une rafale de paquets (de grandeur aléatoire entre 1 et 255). Par conséquent, prenez 1 heure pour bien comprendre comment exécuter ce code du producteur/consommateur sur 2 cores en lisant bien les figures 6 à 18 et le texte qui les accompagne. N'hésitez pas à poser des questions au chargé de laboratoire. Si vous comprenez bien le fonctionnement de ce producteur/consommateur et comment le programmer sur 2 cœurs, le travaille de transférer TaskGenerate sur core 1 sera simple, puisque vous pourrez réutiliser le même découpage. Plus précisément, comme vous aurez créé 2 BSPs (1 pour core 0 et 1 autre pour core 1), vous pourrez reprendre ces 2 BSPs.

Dans ce laboratoire, vous allez poursuivre dans le projet Vivado et SDK de la partie 2. **Mais avant tout faite un backup de cette partie en copiant project_1 dans un autre répertoire (p.e. projet_1_backup).**

Voici dans ce qui suit comment exécuter ces 2 programmes en mode AMP dans votre projet courant, chaque programme sur un cœur avec son propre port de uC/OS-III :

1) Le consommateur sera sur core 0 et il sera ***slave***. Vous devez créer une nouvelle application que vous nommerez *Ex_consommateur* et qui utilisera ucos_bsp_0. Comme à l'habitude ajoutez le *Hello World* de uC/OS-III sur la fenêtre suivante.

2) Cliquez avec le bouton de droite pour ajouter une option sur le setting du BSP (flèche rouge sur Figure 6) i.e. :
    - mettre consommateur comme un slave, i.e. UCOS_AMP_MASTER = false (Figure 7)[5]

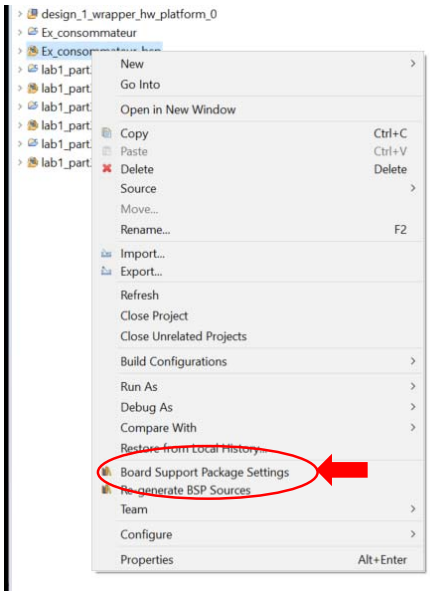


Figure 6

[5] Attention si vous désirez plus tard revenir à un fonctionnement normal du core 0 (comme pour la partie 2), il faudra remettre UCOS_AMP_MASTE à true sur la figure 7.

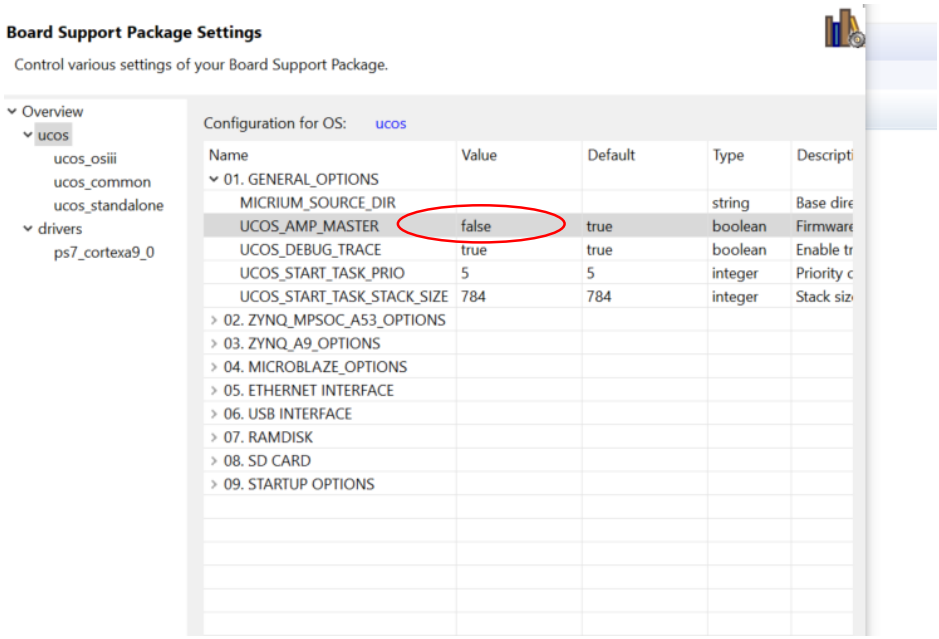


Figure 7

3) Remplacer le fichier *app.c* par *consommateur.c* et modifier le linker script (fichier *lscript.ld*) pour avoir celui de la figure 3.

4) Vous devez ensuite créer un 2e BSP sur core 1 qui sera maître. Faire un File → Board Support Package. Dans le champ Project Name mettre le nom de *ucos_bsp_1*. Puis sélectionner **ps_cortex9_1** et non sur ps_cortex9_0 comme choix de CPU. Finalement, indiquez *ucos* dans le choix du OS. Le résultat est à la figure 8.

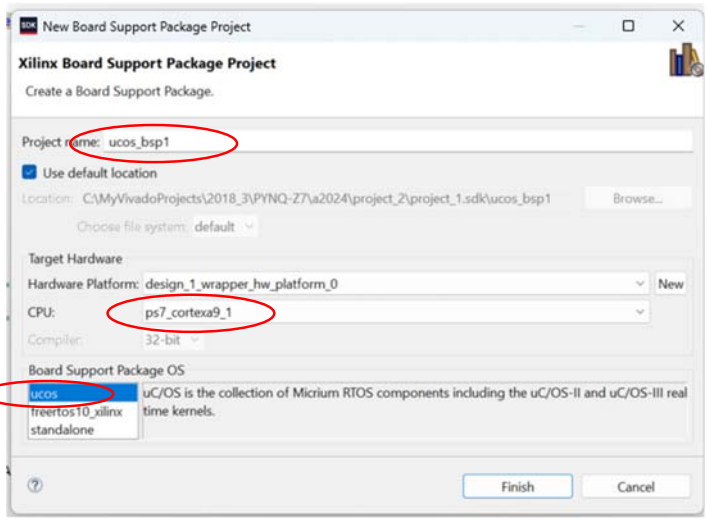


Figure 8

5) Cliquez sur finish et mettre les mêmes paramètres que ucos_bsp_0 c'est-à-dire :

    a. Les drivers avec int 3.8,

    b. ucos_standalone pour stdout ops7_uart_0, et

    c. finalement, mettre les bons paramètres de profondeur de FIFO (OS_CFG_MSG_POOL_SIZE) et de fréquence (OS_CFG_TICK_RATE_HZ) c'est-à-dire 6000 de pool size (fifo) et 8000 Hz pour le tick rate

   mais laisser ucos GENERAL_OPTIONS à UCOS_AMP_MASTER = true car core 1 sera maitre.

6) Ensuite, créez votre application *Ex_producteur* et assurez-vous bien de la faire exécuter sur le CPU **ps_cortex9_1** avec **BSP ucos_bsp_1.** Remplacer le fichier *app.c* par *producteur.c* et modifier le linker script pour avoir celui de la figure 4.

7) Il ne vous reste qu'à programmer votre carte, ce qui implique programmer la partie matérielle (PL) et la partie logiciel (PS avec les ELF) avec la fenêtre de *Debug Configurations* (onglet Run dans la barre horizontale).

    a. La figure 9 montre l'onglet *Target Setup* pour programmer le FPGA (partie PL). **Cette fois, il ne faut pas mettre l'option *Reset entire system***. Il faut laisser la partie PS s'initialiser elle-même.

    b. La partie PS demande deux chargements : la figure 10 montre le chargement de l'application *EX_consommateur.elf* , alors que la figure 11 montre le chargement de *EX_producteur.elf*

    Attention : *Reset processor* reset les 2 cores (vous devez le faire seulement sur core1 et pas sur core0).

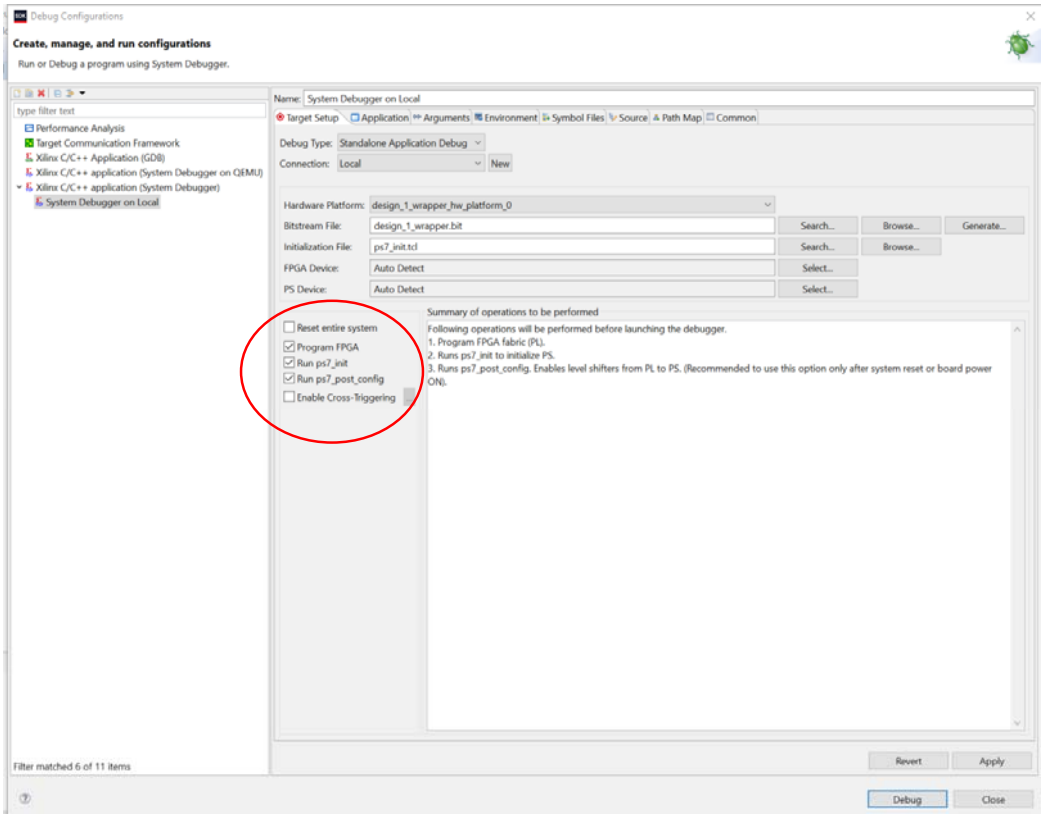


Figure 9 Programmation de PL sans reset

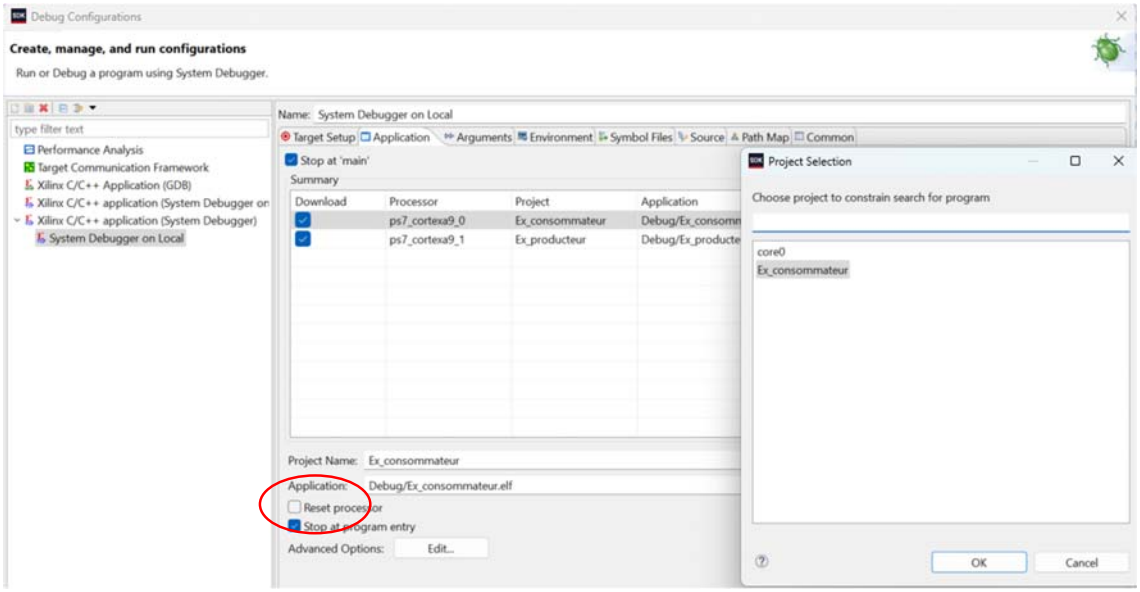


Figure 10 Programmation du core 0 (ELF) sans reset

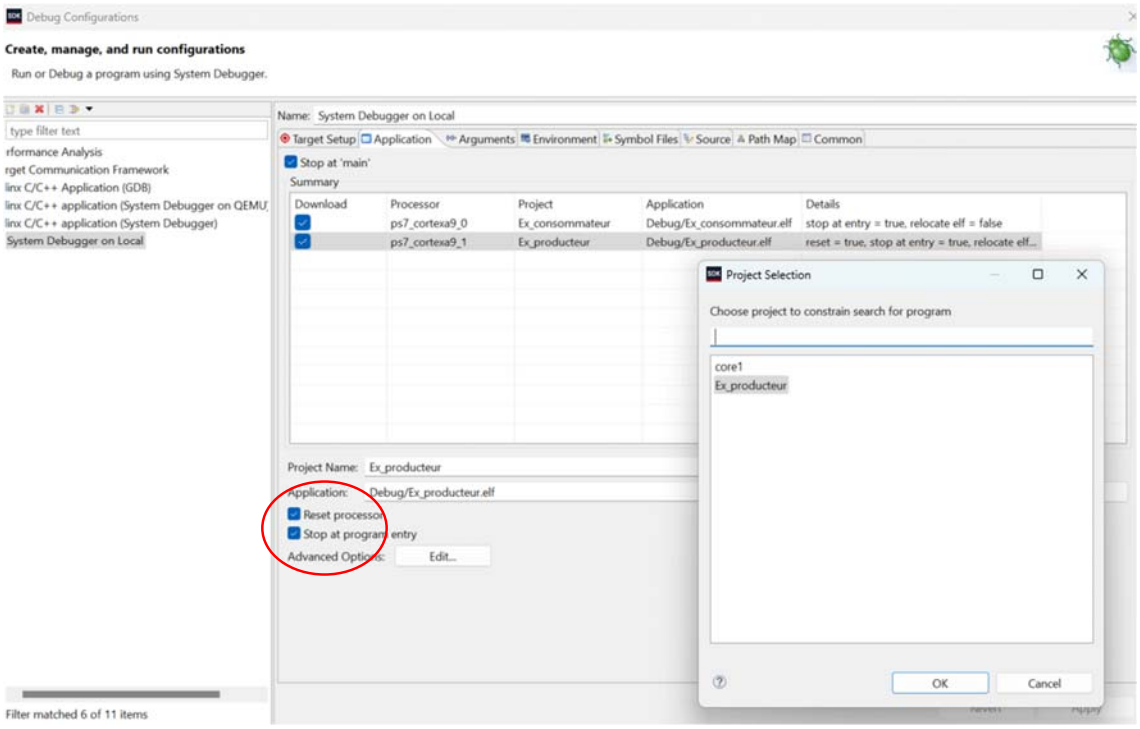


Figure 11 Programmation du core 1 (ELF) mais avec reset cette fois

8) Cliquez ensuite sur *Debug* dans le bas de la fenêtre (voir figure 9).
9) Vous avez maintenant le choix de démarrer soit le core 0 ou le core 1. Comme notre application démarre par le producteur qui fait une mise à 0 des signaux *ack* et *req* (Figure 5), nous allons d'abord démarrer le ARM Cortex-A9 MPCore # 1 en premier. Comme montré à la figure 12 :
    i. cliquez sur ce dernier, puis
    ii. cliquez sur Resume. Le core 1 va alors s'arrêter à la ligne 109. Préparez alors la fenêtre SDK terminal puis faites un autre Resume. Vous devriez voir l'initialisation du core 1 complété dans la fenêtre SDK terminal (Figure 13). Toujours sur la figure 13, vous devriez aussi voir que core 1 est en mode *Running*. Cliquez sur *Suspend* (Figure 14) et vous devriez voir que core 1 a stoppé à la ligne 159. On vous expliquera au laboratoire le protocole mais disons que pour l'instant considérez que core 1 (producteur) est en attente active du signal *ack* (adresse 0x3000014 dans la mémoire partagée). Or ce signal *ack* est en fait le signal *req* du consommateur sur core 0 (voir fig. 5).

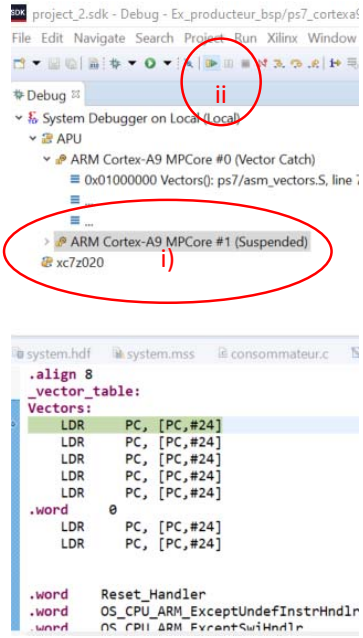


Figure 12

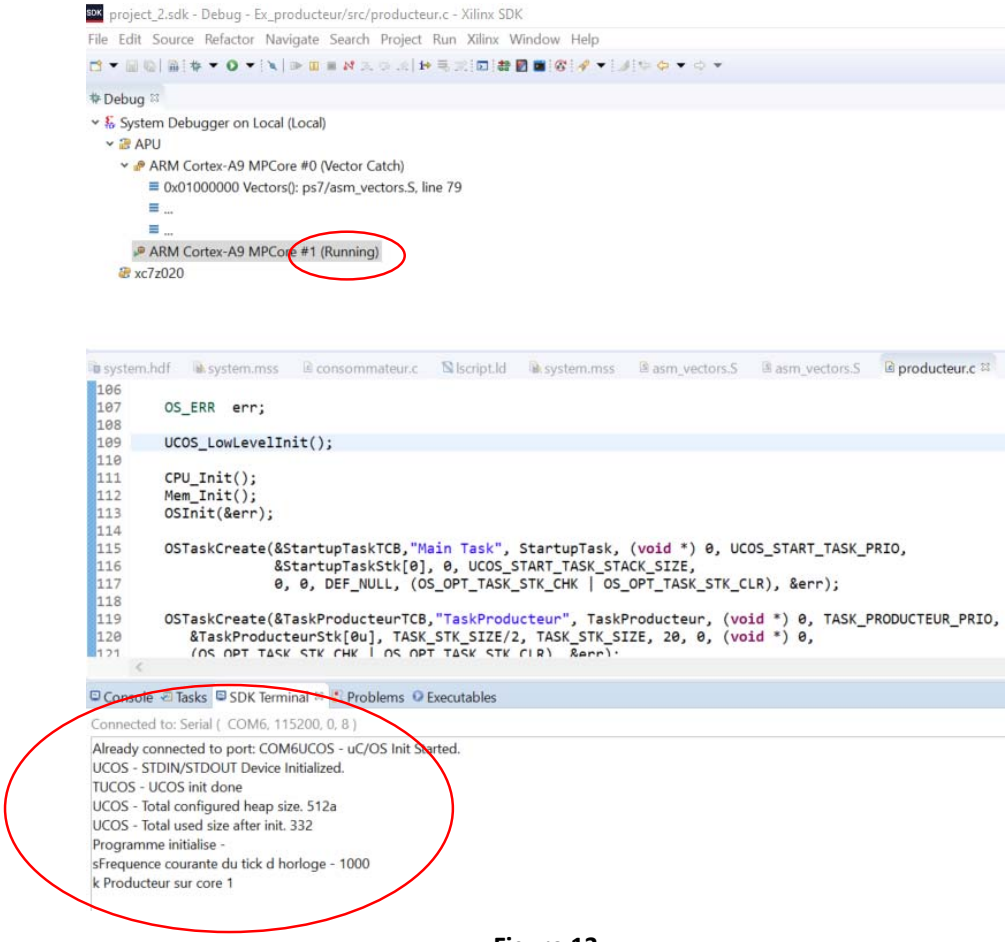


Figure 13

(Remarque suite à l'étape 8)

Notez que lorsque vous faites ainsi un *Suspend*, vous pouvez observer tout le contenu des variables de core 1. Prenez le temps de noter quelque part les adresses et le contenu de *ack* et *req* en mémoire partagée. Laissez pour l'instant core 1 en mode Suspend.

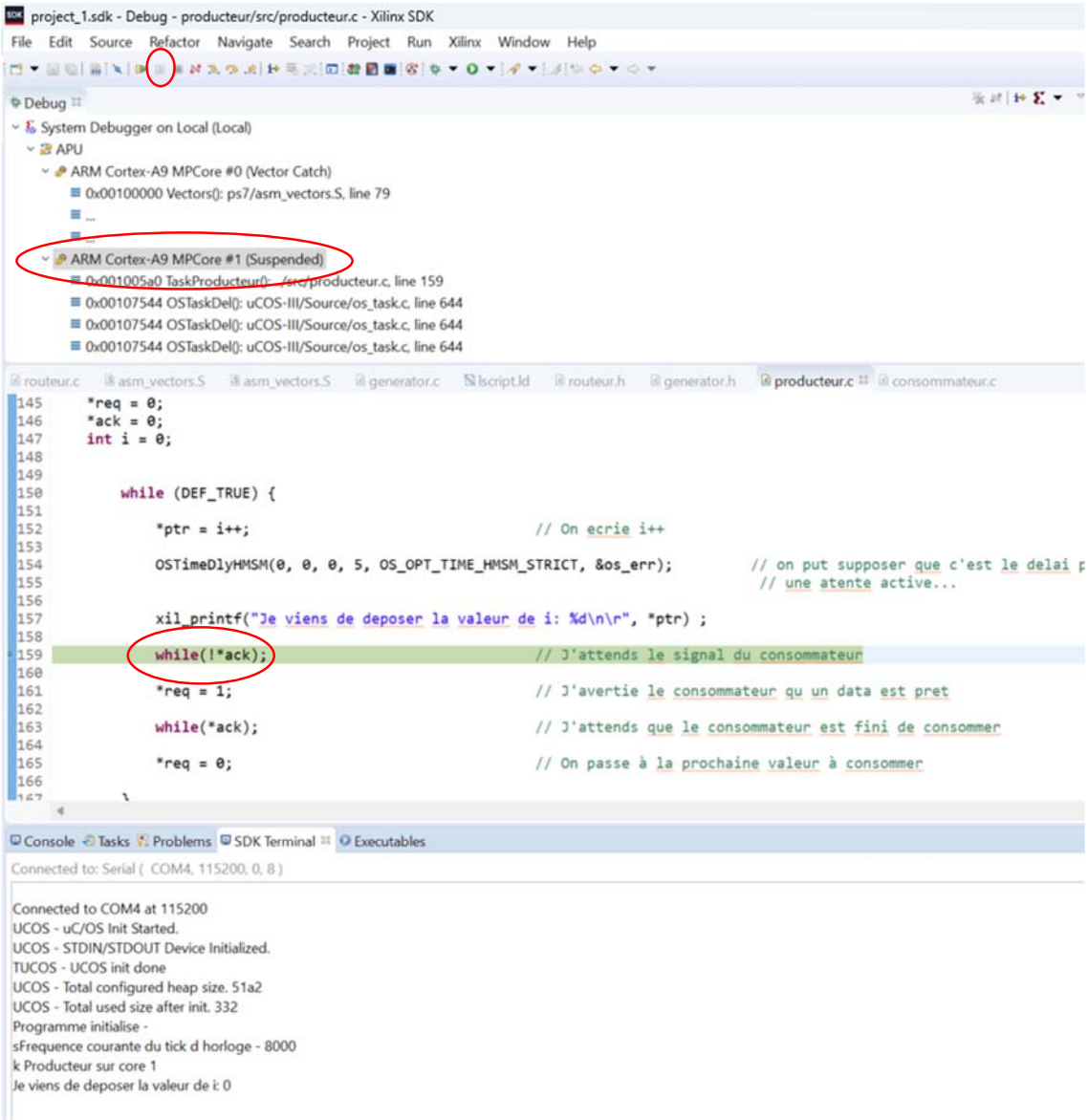


Figure 14

10) Nous allons maintenant démarrer core 0. Procédez comme à l'étape 8 mais en cliquant sur *Cortex-A9 MPCore # 0* puis sur *Resume*. Faites ensuite un deuxième *Resume*. Le système

devrait s'être initialisé puis être en mode Running (Figure 15). À nouveau si vous cliquez sur *Suspend* core 0 va s'arrêter à la ligne 152 lui aussi en attente active du signal *ack*. Vous devriez aussi pouvoir observer que core 0 a mis ***req* à 1** juste avant de se mettre en attente de *ack*. **Bonne nouvelle! Nous ne sommes pas dans un deadlock.**

Remarque : Faire attention, car parfois on croit observer les variables de core 0 mais on observe celles de core 1, si on a par exemple cliqué sur *Cortex-A9 MPCore # 1 Suspended* et non sur *Cortex-A9 MPCore # 0 Suspended*.

11) Redémarrez maintenant core 1 en cliquant *ARM Cortex-A9 MPCore # 1* et cliquez sur *Resume* puis à nouveau sur *Suspend*. Vous verrez que core 1 attend maintenant activement à la ligne 163. Il attend que le consommateur ait consommé (***req* à 0**). Faites un *Resume* sur core 1, puis un *Resume* sur core 0. Votre système AMP devrait maintenant être en fonction!

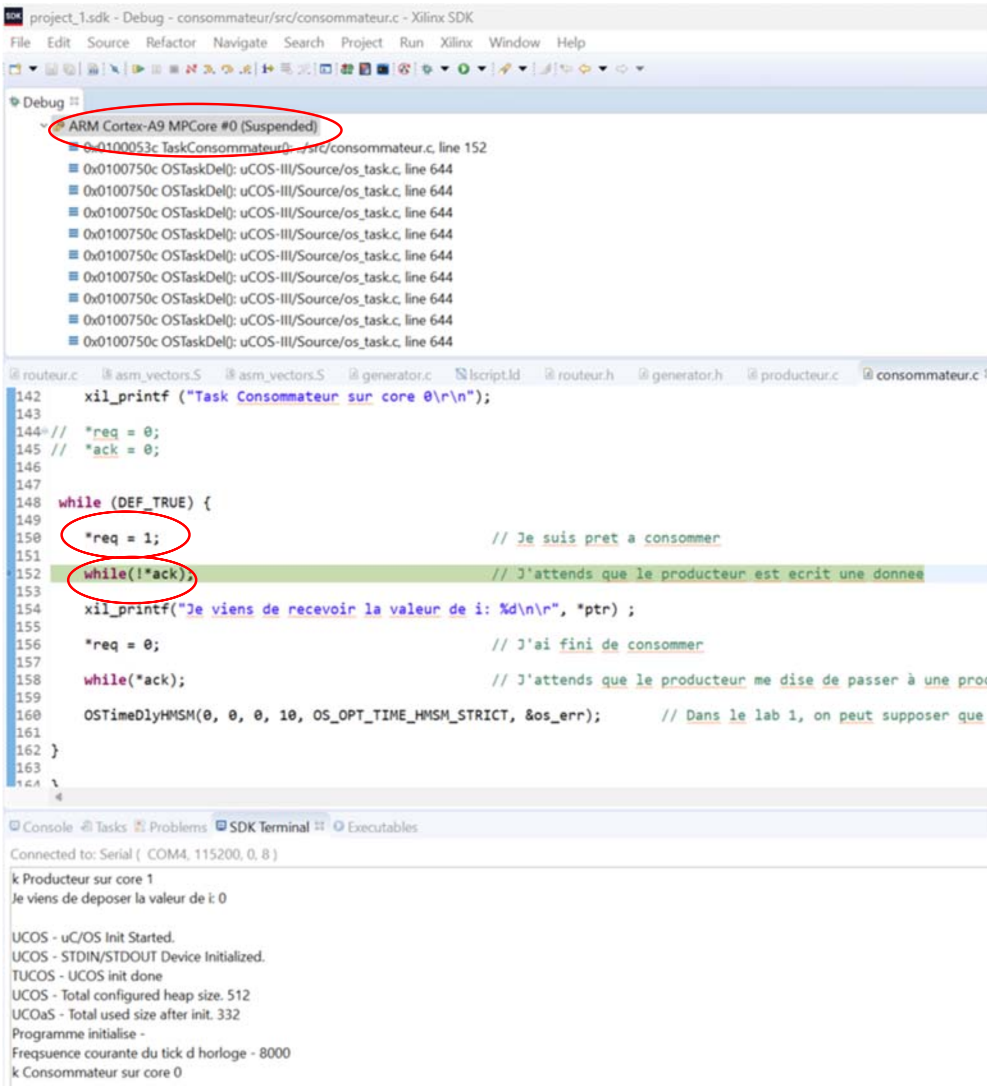
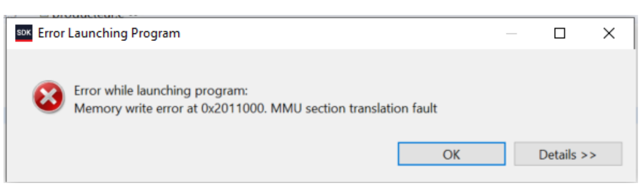


Figure 15

Autres remarques importantes :

1) Entre chaque exécution de Debug Configurations, assurez-vous de faire un reset (voir bouton SRST sur figure 18) sur la carte. Surtout si le message suivant apparait :



Si le bouton SRST crée quand même l'erreur MMU, mettre la switch à OFF et ensuite à ON mais attention vous devrez alors redémarrer le UART du SDK Terminal (ce qui n'est le cas avec SRST).

2) Ne laissez pas votre système fonctionner trop longtemps (plusieurs minutes). Ça engorge le UART du SDK terminal et peut faire planter SDK. Le problème est que la prochaine fois que vous démarrez SDK, la fenêtre de départ ne s'ouvre plus… Pour être certain d'arrêter la trace, utilisez les boutons reset du point 1).

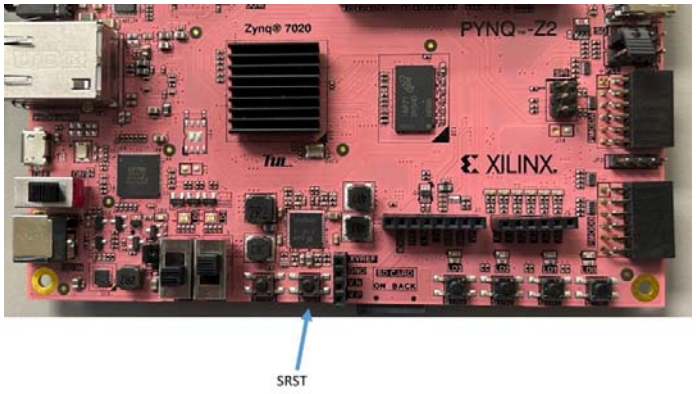


Figure 18

3. Manipulations

Dans ce qui suit, vous devez réaliser 4 manipulations (section 3.1 à 3.4)

3.1) : Déplacer TaskGenerate de core 0 à core 1

Tâche à faire : partitionner le routeur en vous inspirant du producteur/consommateur

Après vous avoir familiarisé avec l'exemple du producteur/consommateur, vous êtes en mesure de compléter une implémentation AMP du routeur selon l'assignation suivante :

- core 0 = tout le système de la séance no 2 moins *TaskGenerate*
  - Créez une application *core0* sur *ps_cortexa9_0* pour laquelle vous allez utiliser le BSP *ucos_bsp_0*.
  - Copier les fichiers du répertoire *src* de votre lab 1 partie 2 dans le *src* de core0.
  - Dans *routeur.c* et *routeur.h*, mettre en commentaire tout ce qui concerne la tâche *Task_Generate*
  - Copier le fichier *lscript.ld* de *src* de consommateur dans le *src* de *core0*.
  - Poursuivre plus bas avec la présentation du pseudo-code du côté de TaskQueuing.

- core 1 = *TaskGenerate*
  - Créez une application *core1* sur *ps_cortexa9_1* p pour laquelle vous allez utiliser le BSP *ucos_bsp_1* créé plus haut.
  - Les fichiers *generateur.h* et *generateur.c*[6] pour cette manipulation sont donnés, mais vous devez bien les comprendre pour les raisons suivantes :
    - Pour compléter la synchronisation avec *TaskQueuing* sur *core0* et
    - éventuellement pour ajouter le code qui permettra l'arrêt (section 3.3).
  - Copier le fichier *lscript.ld* de src de consommateur dans le src de *core0*.

Voici comment sera organisé la DDR vu de core 1 (dans les déclarations de *TaskGenerate*) :

```
// Variables partagees entre core 0 et core 1, au total:
// 4 mots de controle suivi d'au maximum 255 paquets de 16 mots
// Les 2 premiers mots servent à la synchronisation
// Le  3e mot sert a indiquer le no du burst
// Le  4e mot indique le nombre de paquet dans le burst (max de 255)
// Puis vont suivre les paquets (entre 1 et 255 paquets selon la fonction random de TaskGenerate)
// Attention ack de Task Generate doit être jumelé avec req de Task Computing
//        et req de Task Generate doit être jumelé avec ack de Task Computing

volatile uint32_t *req = (uint32_t*)(BASEADDR + 0x00);          // 1er mot: signal comme quoi on est prêt à recevoir un paquet
volatile uint32_t *ack = (uint32_t*)(BASEADDR + 0x04);          // 2e mot: signal comme quoi on est prêt à envoyer un paquet
volatile uint32_t *burst_no = (uint32_t*)(BASEADDR + 0x08);
volatile uint32_t *number_of_packets = (uint32_t*)(BASEADDR + 0x0C);

Packet *ppacket;
ppacket = BASEADDR + 0x10;  // C'est à partir de cet adresse qu'on va lire le 1er  paquet de 16 mots

*req = 0;
*ack = 0;
```

[6] Le code de TaskGenerate est légèrement différent du votre en ce sens qu'il n'utilise pas *isGenPhase* mais détermine tout de suite le nombre de paquet à générer et fait un for. Mais bref la fonctionnalité est identique à votre code.

Voici comment sera organisé la DDR vu de core 0 (dans les déclarations de types et variables de *TaskQueuing*) qui servira à mettre en place le protocole du *handshacking :*

```
// Variables partagees entre core 0 et core 1, au total:
// 4 mots de controle suivi d'au maximum 255 paquets de 16 mots
// Les 2 premiers mots servent à la synchronisation
// Le  3e mot sert a indiquer le no du burst
// Le  4e mot indique le nombre de paquet dans le burst (max de 255)
// Puis vont suivre les paquets (entre 1 et 255 paquets selon la fonction random de TaskGenerate)
// Attention ack de Task Generate doit être jumelé avec req de Task Computing
//        et req de Task Generate doit être jumelé avec ack de Task Computing

volatile uint32_t *req = (uint32_t*)(BASEADDR + 0x04);          // 1er mot: signal comme quoi on est prêt à recevoir un paquet
volatile uint32_t *ack = (uint32_t*)(BASEADDR + 0x00);          // 2e mot: signal comme quoi on est prêt à envoyer un paquet
volatile uint32_t *burst_no = (uint32_t*)(BASEADDR + 0x08);
volatile uint32_t *number_of_packets = (uint32_t*)(BASEADDR + 0x0C);

Packet *ppacket;
ppacket = BASEADDR + 0x10;  // C'est à partir de cet adresse qu'on va lire le 1er  paquet de 16 mots

*req = 0;
*ack = 0;
```

Voici le pseudo-code du côté de *TaskQueuing* (core 0):

1) On vérifie avec le flag que le système est bien initialisé et démarré via BP0 (Fig. 2 de la partie 2)
2) *TaskQueuing* indique avec req = 1 qu'il est prêt à consommer une rafale de paquet de *TaskGenerate* situé maintenant sur core 1.
3) On attend de manière active avec *while(!*ack)* que *TaskGenerate* ait envoyé une rafale.
   À partir de 4) on va consommer la rafale.
4) ppacket = BASEADDR + 0x10; // Toujours partir une rafale à 0x10
5) Ici plutôt que de lire dans le fifo de *TaskQueueing* avec *OSTaskQPend*, on va aller lire la rafale produite dans la DDR par TaskGenerate :
   Pour i = 1 jusqu'à **number_of_packets* lire la rafale complète de la mémoire partagée :

```
for (int i = 1; i <= *number_of_packets; ++i) {

    /* On lit le paquet dans la DDR */
    OSSemPend(&Sem_MemBlock,0, OS_OPT_PEND_BLOCKING, &ts, &err);
    Packet *packet = (Packet *)OSMemGet(&BLockMem, &err);
    packet->src = (ppacket->src);
    packet->dst = (ppacket->dst);
    packet->type = (ppacket->type);
    packet->crc = (ppacket->crc);
    packet->timestamp = CPU_TS_Get64();

    for (int i = 0; i < ARRAY_SIZE(packet->data); ++i) {
        packet->data[i] = (ppacket->data[i]);
    }
```

6) Bien que vous ayez copié le CRC nous n'allons pas le traiter, car j'ai remarqué que suite au transfert via la DDR plusieurs CRC semblent se corrompre. Pour vous éviter des problèmes, nous allons mettre en commentaire le test :
   if (computeCRC((uint16_t)packet, sizeof(*packet)) != 0)*
   Vous pouvez remplacer par *if (0 != 0)* afin de toujours faire le *else*…

7) De sorte que le else va envoyer dans le fifo de *TaskComputing* selon la priorité du paquet. Puis on passe au prochain paquet de la rafale dans la DDR avec

```
ppacket++; // +17
```

8) Une fois tous les paquets d'une rafale traiter on indique qu'on a fini de consommer avec *req = 0* et qu'on pourra passer à une autre valeur.
9) On attend de manière active avec *while(*ack)* que *TaskGenerate* est également prêt à passer à la prochaine rafale.
10) *TaskQueuing* fait une pause avec un délai suffisant pour vider les fifos et pendant ce temps *TaskGenerate* peut préparer (en **parallèle**) la prochaine rafale.

Exécutez votre code en suivant les mêmes étapes de démarrage de Debug Configuration) étapes 7 à 11 de la section 2). Attention! Si vous avez laissé la variable *delai_pour_vider_les_fifos_msec* à la valeur trouvée au lab 1 partie (question 3), il est fort possible que les délais de pire temps video, audio et autres soient plus élevés que celles obtenues au lab 1 et que les fifos montent rapidement à 1024… Nous allons examiner ce problème à la prochaine section.

3.2) Analyse performance avec 2 cores plutôt qu'un seul

Cette section est assez courte, mais il faut bien y penser (point 2). Donc 2 choses :

1) Si vous avez rencontré le problème de la variable *delai_pour_vider_les_fifos_msec* assignée à une valeur trop courte (dernier paragraphe de la section 3.1), augmenter par une valeur 5 à 10 fois plus grande que celle obtenue à la question 3 de la partie 1. En principe, si vous recompiler et exécuter, vous devriez rencontrer les mêmes valeurs que pour les résultats du lab 1 partie 1 . **Vous pouvez répondre à la question no 1.**

2) En principe le fait d'avoir déplacé *TaskGenerate* sur le core1 devrait permettre aux tâches restantes sur le core 0 (dont *TaskQueuing*) de s'exécuter plus rapidement. Pourtant, l'augmentation de la variable *delai_pour_vider_les_fifos_msec* semble indiquer le contraire. Pourquoi selon vous? **Répondez à cette question qui fait l'objet de la question no 2.**

3.3) Arrêt de core 1 avec watchdog

Il s'agit ici d'ajouter un watchdog à TaskGenerate pour mettre fin à l'exécution sur core 1 quand on appuie sur BP2 (fig. 1 partie2). Nous avons vu en classe le rôle et l'utilisation du watchdog timer dans uC/OS-III (voir aussi sur Moodle, l'exemple de code minuterie watchdog dans la section Ressource et codes). L'idée ici est de démarrer un watchdog timer s'il s'écoule plus de 30 secondes entre 2 signaux req de TaskComputing. Ce qui veut probablement dire que cette dernière est en panne. Donc si après 30 secondes rien ne sait produit, une fonction de callback devra mettre proprement fin au programme de core 1 (tout comme vous avez fait dans la partie 2, de manière propre…). Je vous laisse choisir le mode de fonctionnement du watchdog (périodique vs one shot). La fonction de rappel portera le nom sera *void ShutdownTaskGenerateFct(OS_TMR *p_tmr, void *p_arg)*.

Attention! N'oubliez pas d'ajuster la priorité de la fonction de callback du watchdog (OS_CFG_TMR_TASK) dans *Board Support Package Settings* de *ucos_bsp_1*, plus précisément dans ucos_iii → APPLICATION. En effet, une valeur trop basse comme 61 peut faire en sorte que la fonction de callback sera continuellement préemptée…

Vous pouvez répondre à la question no 3.

3.4) Terminaison propre

Suite à la réalisation 3.3, il faudra faire une terminaison que je nommerais *propre*. Ça veut dire ici :

1) Vider les queues en rendant la mémoire (free)

   Ici je vois deux manières de faire :

   - en utilisant les fonctions uC/OS-III pertinente ou
   - encore de manière beaucoup plus simple avec :

   OSTimeDlyHMSM(0, 0, delai_pour_vider_les_fifos_sec, delai_pour_vider_les_fifos_msec, OS_OPT_TIME_HMSM_STRICT, &err);

   En choisissant un délai suffisamment long pour vider toutes les queues…
2) Créer une fonction *delete_events* qui va détruire ce qui a été créé dans *create_events* tout en utilisant les fonctions uC/OS-III pertinente.
3) Détruire toutes les tâches en utilisant les fonctions uC/OS-III pertinente.

5. Question pour le rapport

Question 1)

Soit les 2 cas suivants qui abstrait le fonctionnement de TaskGenerate :

Cas 1)

- Generation d'une rafale (burst)
- while(!*ack);
- *req = 1;
- xil_printf("Task_Generate Fin rafale no: %d completee avec %d paquets\n", *burst_no, *number_of_packets);
- while(*ack);
- *req = 0;
- burst_number++;

Cas 2)

- while(!*ack);
- *req = 1;
- Génération d'une rafale
- while(*ack);
- *req = 0;
- burst_number++;
- xil_printf("Task_Generate Fin rafale no: %d completee avec %d paquets\n", *burst_no, *number_of_packets);

Lequel des cas s'applique à votre implémentation et quel est son avantage par rapport à l'autre cas.

Question 2)

En principe le fait d'avoir déplacé TaskGenerate sur le core1 devrait permettre aux tâches restantes sur le core 0 (dont TaskQueuing) de s'exécuter plus rapidement et donc d'améliorer les pires temps video, audio et autres. Pourtant, l'augmentation de la variable *delai_pour_vider_les_fifos_msec* pour respecter le 50% de remplissage des fifos lors de la manipulation 3.1 semble indiquer le contraire. Pourquoi selon vous? Expliquez en détails.

N.B. Si parfois vous n'avez pas eu besoin d'augmenter la variable *delai_pour_vider_les_fifos_msec* et que vous êtes convaincu de son minimum, expliquez en ce sens.

Question 3) À propos de la minuterie watchdog

a) Expliquez comment vous avez implémenté votre watchdog. Plus précisément, expliquez les paramètres de OSTmrCreate, justifiez le mode de fonctionnement du watchdog et décrire comment la fonction de callback réalise le travail de stopper TaskGenerate.

b) J'ai dit en classe qu'une des limitations du watchdog est sa précision limitée à un multiple du tick d'hologe (tick rate). Selon vous, la solution offerte par Xilinx avec le composant *AXI Timerbase Watchdog Timer* de la librairie offrirait-elle un comportement similaire au watchdog utilisé à la section 3.3 mais avec plus de précision ? Justifiez.

Références :

https://docs.amd.com/v/u/en-US/pg128-axi-timebase-wdt et https://github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/wdttb/examples/xwdttb_intr_example.c

6. Barême de correction

Questions	**Notes**	**Commentaires**
1	1pt	
2	1 pt	
3	1 pt	
Fonctionnement	5 pts	
Total	8 pts	

Guy Bois
Responsable du cours

Date de remise du rapport :

Gr 2 : 5 novembre 2024 au plus tard 23h59

Gr 1 : 12 novembre 2024 au plus tard 23h59

Gr 3 : 14 novembre 2024 au plus tard 23h59