M2LSE : projet UE SoC
Détection de dépassement de temps d'exécution

Stéphane Rubini

Ce document décrit le projet à réaliser dans le cadre de l'UE System-On-Chip.

En ordonnancement temps-réel dur, on budgétise un temps de calcul processeur pour chaque job des tâches qui s'exécutent sur le système. Ce budget est déterminé par le pire temps d'exécution d'un job (WCET pour Worst Case Execution Time). Cette approche permet, sous certaines conditions, de vérifier que le processeur dispose de suffisamment de puissance de calcul pour garantir les contraintes de temps de l'ensemble du système (*deadline*).

Les événements pertinents du point de l'analyse d'ordonnancement sont le début, la fin, la suspension et la reprise d'un job. Le cumul des intervalles de temps délimités par ces événements doit rester inférieur au WCET budgétisé pour un job.

Le but du projet est de développer un moniteur matériel qui comptabilise pour chaque job le temps d'occupation du processeur. Si le temps dépasse le WCET, une interruption IRQ est levée qui informe le processeur de la détection d'une anomalie.

Étapes du projet

1. Concevoir le module matériel, et coder son comportement en VHDL. La capacité de moniteur sera limitée à un nombre de tâches maximum donné.

   Quelques problèmes à résoudre dans cette phase de conception :
   - Comment représenter le temps dans le circuit (durée maximum, précision) ?
   - Quels sont les événements à signaler au circuit ? Comment les représenter (les coder) ?
   - Comment stocker les *WCET* et les temps d'exécution des tâches dans le circuit ?
   - Comment mettre à jour le temps d'exécution des tâches, en fonction du temps et des événements d'ordonnancement ?
   - Comment détecter qu'un temps d'exécution dépasse le WCET ?
   - ...

Simuler le module que vous avez conçu, le *testbench* produisant les événements d'ordonnancement selon un scénario temporel prédéfini. À cette étape, l'outil GHDL ou Vivado pourront être utilisés.

2. Intégrer le module dans un périphérique AXI esclave qui sera implanté sur un PSoC Zynq.
3. Développer le pilote associé, et tester le moniteur en simulant un ordonnancement sur le processeur ARM.

Modalités d'évaluation Le compte-rendu du projet devra être remis **sous forme papier** avant le 10/01/2021. Le travail peut être effectué individuellement ou en binôme.

Le compte-rendu présentera notamment :

1. le travail effectué,
2. l'architecture du module matériel développé,
3. le diagramme Vivado "Block Design",
4. la méthode de test du module et du système,
5. les codes VHDL et C écrits (c'est à dire non générés automatiquement par Vivado).

Remarque : une démonstration sur machine du projet pourra âtre demandée par l'enseignant à la discrétion de celui-ci.

Complément : gestion des interruptions

La plupart des périphériques disponibles dans le PSoC Zynq sont en mesure d'envoyer un signal d'interruption au processeur. En réponse à une interruption, le processeur peut déclencher l'exécution d'une fonction (ou routine) destinée à traiter la situation qui a causé l'interruption. La gestion des IT nécessite une configuration relativement complexe au niveau du processeur et de l'OS, destinée à associer le signal et la routine d'interruption et à définir les moments où l'interruption est prise en compte.

Les processeurs ARM Cortex-A9 qui composent le PSoC Zynq reçoivent 2 signaux d'interruption nFIQ et nIRQ, et des signaux *reset*.

L'initialisation du système de gestion d'interruption du Cortex-A9 est prise en charge par les fonctions dont le nom commence par l'extension `Xil_Exception`.

Le traitement logiciel des interruptions et des exceptions consiste à dérouter le flot d'exécution du programme vers une fonction (routine d'interruption). Une table de vecteurs d'interruption est positionnée à partir de l'adresse 0 en mémoire et définit les adresses des routines en fonction du type d'interruption. 7 types d'interruption sont traités par l'architecture ARM

1. *Reset* : remise à 0 du système déclenché par les signaux physiques de RESET
2. *Undefined instruction* : le processeur ne reconnaît pas l'instruction qu'il doit exécuter
3. *Software Interrupt (SWI)* : exception générée par logiciel en mode utilisateur, qui permet typiquement d'entrer en mode superviseur (appel système)
4. *PrefetchAbort* : tentative de chargement d'une instruction à partir d'une adresse illégale.
5. *Data Abort* : tentative d'accès à une donnée enregistrée à une adresse illégale
6. IRQ : réception d'une interruption matérielle sur la ligne IRQ
7. FIQ : réception d'une interruption matérielle sur la ligne FIQ

La position des vecteurs d'interruption dans la table est codée par les symboles pré-processeur suivant :

```
#define XIL_EXCEPTION_ID_RESET                  0
#define XIL_EXCEPTION_ID_UNDEFINED_INT          1
#define XIL_EXCEPTION_ID_SWI_INT                2
#define XIL_EXCEPTION_ID_PREFETCH_ABORT_INT     3
#define XIL_EXCEPTION_ID_DATA_ABORT_INT         4
#define XIL_EXCEPTION_ID_IRQ_INT                5
#define XIL_EXCEPTION_ID_FIQ_INT                6
```

Les 2 fonctions suivantes permettent d'associer, ou de dissocier, une fonction (`handler`) à un type d'interruptions. L'argument *id* identifie le vecteur concerné dans la table. Une donnée peut être transmise à cette routine si-nécessaire (champs `data`).

```
typedef void (*Xil_ExceptionHandler)(void *data);
extern void Xil_ExceptionRegisterHandler(
                    unsigned int id,
                    Xil_ExceptionHandler handler,
                    void *data);
extern void Xil_ExceptionRemoveHandler(unsigned int id);
r
```

Les interruptions reçues sur les lignes IRQ et FIQ peuvent, ou non, être prises en compte selon la configuration du processeur. Les macro-instructions suivantes contrôlent la masquage des signaux d'interruptions.

```
#define XIL_EXCEPTION_FIQ           XREG_CPSR_FIQ_ENABLE
#define XIL_EXCEPTION_IRQ           XREG_CPSR_IRQ_ENABLE
#define Xil_ExceptionEnableMask(mask)
#define Xil_ExceptionDisableMask(mask)
```

Generic Interrupt Controller Un périphérique appelé GIC pour *Generic Interrupt Controller* est défini par ARM et complète ce modèle matériel initial. Il fait partie d'un *processing system* du composant *Zynq*. La figure 1 montre l'architecture simplifiée du système.

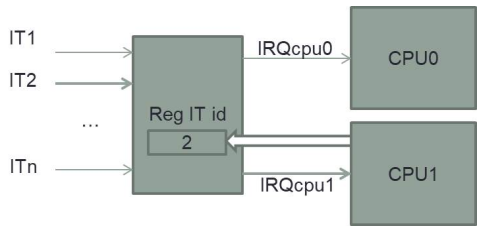

FIGURE 1 – Interactions simplifiées entre le GIC et les processeurs

Les 2 fonctions suivantes initialisent le GIC et retourne un "descripteur" d'accès à ce périphérique de type `XSucGic`. La configuration contient notamment une table supplémentaire de vecteurs d'interruption.

```
XScuGic_Config * XScuGic_LookupConfig(device_id);
int  XScuGic_CfgInitialize(XScuGic * config,
        XScuGic_Config *config_hw,
        unsigned int CpuBaseAddress);

typedef struct {
        Xil_InterruptHandler Handler;
        void *CallBackRef;
} XScuGic_VectorTableEntry;
typedef void (*Xil_InterruptHandler)(void *data);
```

Le GIC concentre plusieurs dizaines de sources d'interruptions provenant des coupleurs de périphériques du *Zynq* ou du FPGA, et les diffuse vers le processeur selon leur priorité respective. La routine d'interruption `XScuGic_InterruptHandler` lit le numéro de l'interruption source dans le GIC, puis déclenche l'exécution d'une routine de "2^{nd} niveau".

La fonction `XScuGic_Connect` installe une nouvelle routine de 2^{nd} niveau dans la table des vecteurs d'interruptions. Le numéro de vecteur `Int_Id` dépend de la mise en œuvre matérielle du GIC. *Vivado* exporte l'identifiant du vecteur par le symbole pré-processeur $XPAR\_FABRIC\_ < nomIP > \_VEC\_ID$. Par exemple, pour l'entrée d'interruption `Core0_nIRQ`, ce numéro sera 31. Enfin, la fonction `XScuGic_Enable` valide la prise en compte d'une interruption.

```
int XScuGic_Connect(XScuGic *InstancePtr,
                    unsigned int Int_Id,
                    Xil_InterruptHandler Handler,
```

```
                        void *CallBackRef);
void XScuGic_Enable(XScuGic *InstancePtr,
                        unsigned int Int_Id);
```

Résumé des opérations de gestion des interruptions

1. Une interruption est reçue en entrée du GIC.
2. Le GIC écrit le numéro de l'interruption ($XPAR\_FABRIC\_ < nomIP > \_VEC\_ID$) dans un registre interne.
3. Le GIC envoie une interruption au processeur concerné par l'interruption initiale.
4. Le processeur exécute la routine d'interruption (de type `Xil_ExceptionHandler`). La routine effectue alors les opérations suivantes :
   — lecture du numéro de l'interruption initiale dans le registre interne du GIC,
   — exécution d'une routine d'interruption de 2^{nd} niveau associée au numéro obtenu (de type `Xil_InterruptHandler`),
   — acquittement de l'interruption dans le GIC.