M1 SESI 2017-2018
Architecture Multi-Processeurs
TP6 : Interruptions vectorisées
Communications avec les périphériques

Kevin Mambu

March 23, 2018

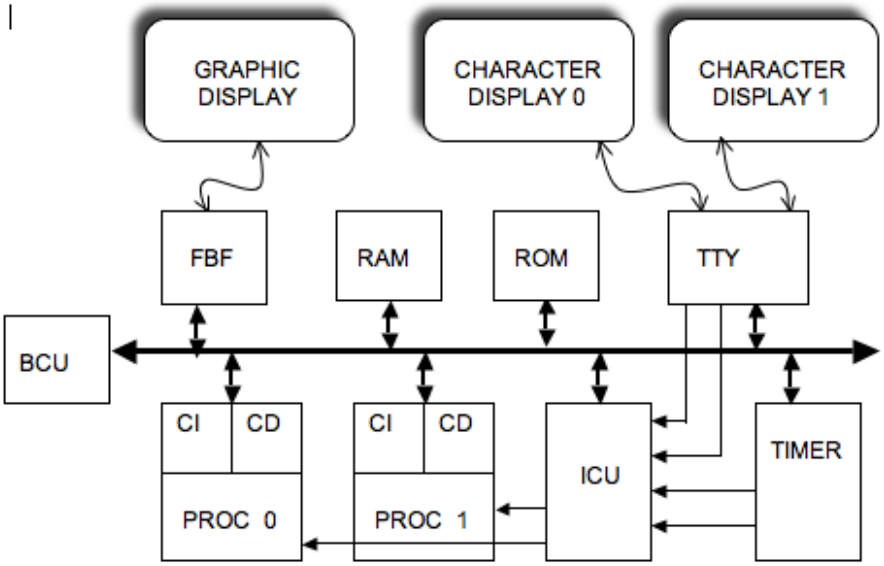
A) Objectifs

Le but de ce TP est d'analyser les mécanismes de communication par interruptions entre les périphériques et le système d'exploitation. Dans une première partie de ce TP, on illustre sur une architecture bi-processeurs le mécanisme des interruptions vectorisées en utilisant un Timer programmable, capable de générer des interrutions périodiques. Dans une seconde partie, on analyse en détail le mécanisme permettant à un programme de lire des catactères à partir d'un terminal TTY.
On dit qu'un périphérique est "mappé" en mémoire lorsqu'il possède des registres adressables par le logiciel (au moyen d'instructions de lecture ou d'écriture du type lw ou sw).

- Les registres accessibles en écriture permettent au système d'exploitation de configurer les périphériques, ou de leur envoyer des commandes.
- Les registres accessibles en lecture permettent au système d'exploitation d'obtenir des informations sur l'état du périphérique.

Pour communiquer avec le système d'exploitation, les périphériques utilisent des interruptions : Une interruption, ou IRQ (Interrupt ReQuest) est un signal booleen actif à l'état haut, qui permet à à un périphérique de "voler " quelques cycles à un processeur pour exécuter une ISR (Interrupt Service Routine). Ces ISR ont généralement pour rôle d'écrire dans des tampons mémoire appartenant au système d'exploitation. Ces tampons de communication sont spécifiques pour chaque type de périphérique.
L'architecture matérielle est identique à l'architecture matérielle définie pour le TP5, mais on utilisera seulement deux processeurs.

B) Architecture matérielle



C) Composants périphériques

Le composant matériel PibusIcu, est un concentrateur d'interruptions vectorisée. Ce composant est souvent appelé PIC (Programmable Interrupt Controller) dans les PCs. Le concentrateur d'interruptions utilisé ici peut concentrer jusque 32 lignes d'interruptions IRQ_IN[I] en entrée (provenant de différents périphériques), vers une seule ligne d'interruption en sortie IRQ_OUT (connectée à un processeur). Il contient un grand "OU" cablé: Il suffit qu'une seule entrée IRQ_IN[i] soit active (état haut) et non masquée, pour que IRQ_OUT passe à l'état haut.
Le composant ICU utilisé contient un encodeur de priorité qui implémente un mécanisme de priorité fixe : si plusieurs lignes d'interruption entrantes IRQ_IN[i] sont actives simultanément, le registre adressable IT_VECTOR contient l'index de l'interruption active qui a l'index le plus petit. Enfin, ce composant permet au logiciel de masquer individuellement chacune des 32 lignes d'interruption entrantes, en écrivant un mot de 32 bits dans le registre IT_MASK du composant PibusIcu.
Le composant PibusIcu est un périphérique multi-canaux: Quand il y a plusieurs processeurs P[k], le contrôleur d'interruptions possède autant de sorties IRQ_OUT[k] qu'il y a de processeurs P[k]. Chaque canal [k] correspond à une sortie IRQ_OUT[k], et se comporte comme un concentrateur d'interruptions indépendant. La seule chose partagée par les différents canaux sont les 32 signaux entrants IRQ_IN[i]. Si il y a plusieurs processeurs, le composant PibusIcu contient un registre de masque spécifique pour chaque canal, ce qui permet au système d'exploitation de décider, pour chaque interruption IRQ_IN[i], à quel processeur elle va être transmise.
Le composant matériel PibusMultiTimer est également un périphérique multi-canaux. Il contient plusieurs timers programmables. Chaque timer a pour fonction de générer des interruptions périodiques, programmables par logiciel. Chaque timer possède sa propre ligne d'interruption. Le code exécuté en cas d'interruption générée par le timer est défini par la routine de traitement de l'interruption _isr_timer (ISR signifie Interrupt Routine Service).
Le composant matériel PibusMultiTty a déjà été utilisé dans les TPs précédents. Il contrôle plusieurs terminaux TTY indépendants. Chaque terminal possède une ligne d'interruption qui lui permet de signaler qu'un caractère a été saisi au clavier. Cette interruption peut être utilisée par le système, lorsqu'on ne souhaite pas utiliser un mécanisme de scrutation pour acquérir les caractères du clavier. Le code exécuté en cas d'interruption générée par le TTY est défini par la routine de traitement de l'interruption _isr_tty_get.
Les spécifications sont en annexe

Question C1

Le composant **PibusMultiTimer** est une cible et non un maître sur le bus parce que :

- Elle n'émet pas de données sur le bus indépendamment d'une requête de la cible.
- Toute emission d'interruption (TIMER_IRQ) est en réalité à la demande du système d'exploitation.

PibusMultiTimer est d'avantage un contrôleur de timers indépendants : `ntimer` est le nombre de timers indépendants sous **PibusMultiTimer**.
Regarder l'annexe des spécifications pour les registres adressables et leurs offsets par rapport à l'adresse de base du segment SEG_TIMER_BASE

Question C2

Le composant **PibusMultiTimer** est une cible et non un maître sur le bus parce que :

- Elle n'émet pas de données sur le bus indépendamment d'une requête de la cible.
- Même si à la suite de la sortie de l'ICU, `IRQ_OUT`, un accès mémoire est fait (lecture/écriture de registres), cela est la conséquence de l'ISR associée à l'interruption et exécutée par le processeur.
- Et elle est également dépendante du système d'exploitation.

`nirq` est le nombre de signaux d'interruptions branchés sur l'ICU (cela indique par conséquence le nombre de périphériques sur l'architecture).
L'ICU du PIBUS est un contrôleur multi-canaux, `nproc` indique le nombre de processeurs reliés à l'ICU et ainsi le nombre de sorties IRQ_OUT.
L'ICU est un contrôleur reprogrammable. Lors du boot, le code du reset a parmi ses tâches d'effectuer le routage interne de l'ICU des requêtes entrantes IRQ_IN[`nirq`] vers IRQ_OUT[`nproc`]. Le système d'exploitation peut également impacter sur le routage interne.
Soit n l'identifiant du processeur au sein de l'architecture vis-à-vis de l'ICU et l'adresse de base du segment SEG_ICU_BASE, chaque processeur est associé à un sous-segment dont l'adresse de base est $SEG\_ICU\_BASE + n \times 5$. Les offsets et les fonctionnalités de chaque registe sont décrits dans l'annexe.

Question C3

L'adresse de base de l'ICU SEG_ICU_BASE doit être alignée multiple de 32×8 octets parce que chaque proceseur lui est attribué 5 registres, soit 20 octets, qu'on rehausse à 32 par souci d'alignement. Relâcher cette contrainte aurait pour conséquence de limiter le nombre de sous-canaux correctement utilisables, car l'un d'entre eux n'aurait pas de mapping mémoire suffisant pour ses registres.

Question C4

Rappelons que nous sommes sur une architecture bi-processeurs (`NPROC` $= 2$).

- IRQ_IN[0] $\rightarrow$ DMA
- IRQ_IN[1] $\rightarrow$ IOC
- IRQ_IN[2+2i] $\rightarrow$ TIMER[i]
- IRQ_IN[3+3i] $\rightarrow$ TTY[i]

D) Lancement des tâches

Question D2

main_prime	*main_pgcd*
`0x004012dc`	`0x004013f0`

Question D3

Le flag de GCC, `freestanding` permet d'assumer que le début du programme principal ne sera pas nécessairement à l'adresse `main`. L'option `no-gpopt` demande à GCC de ne pas utiliser de pointeur global vers le segment seg_data. De ce fait, GCC utilise le mécanisme d'accès de labélisation des données, ce qui nous fait notre table de saut.

Question D4

Le programme ne peut pas notifier de l'entrée d'un caractère car le processeur n'est pas connecté à l'ICU et donc ne peut pas acquitter de l'interruption.

E) Activation du timer

On veut maintenant activer les interruptions provenant du TIMER. On rappelle qu'à chacune des lignes d'interruption est associée une routine d'interruption (ISR ou Interrupt Service Routine) qui est spécifique au périphérique qui a généré l'interruption, et qui est exécutée par le processeur lorsque les interruptions ne sont pas masquées. Activer les interruptions du TIMER est donc à équivalent à lancer sur chacun des 2 processeurs de l'architecture une tâche de fond, consistant à exécuter périodiquement l'ISR _isr_timer.

Question E1

Pour se brancher à l'ISR de l'interruption concernée i, le processeur passe par le tableau interrupt_vector, qui est une table de saut, puis saute à l'adresse stockée à interrupt_vector[i].
Séquence d'appels de fonctions jusqu'à isr_timer :

- Point d'entrée au GIET
- `r27 <= Cause register`
- `r26 <= _cause_vector`
- `r26 <= &_cause_vector`

$$XCODE$$

- `r26 <= _cause_vector[XCODE]`
- Branchement à l'adresse dans `r26` (_int_handler)
- Réservation d'espace sur la pile d'appel
- Sauvegarde de contexte (`r1-r31, HI, LO, EPC`)
- Brancchement à la fonction C `_int_demux`, de irq_handler.c
- Lecture du registre de l'ICU _ICU_IT_VECTOR
- S'il n'y a pas d'interruption active (lecture retourne 32) → retour à _irq_handler, sinon...
- `isr <= _interrupt_vector[index]` (_isr_timer)
- Branchement à la routine _isr_timer

Question E2

_isr_timer gère les deux timers connectés respectivement aux processeurs 0 et 1. Plus précisément, pour un timer TIMER[id], il acquitte l'interruption du timer puis affiche un message sur le TTY.

Question E3