Nom : AIDER
Prénom : Smail
N°Etudiant : 3603379
Parcours : SAR
Responsable : M Alain Greiner

Compte-Rendu TP8
Contrôleur de disque & partage de périphériques

B) *Contrôleur de disque*

Question B1

Signification des arguments du constructeur du composant PibusBlockDevice :
- block_size : la taille d'un bloc disque en nombre d'octets (512 par défaut)
- latency : représente le délai d'attente (en nombre de cycles) entre la réception d'une commande et l'obtention de la donnée (voir TP01 - Section C).

Question B2

Une image à pour taille : 128 * 128 octets
→ Une image occupe (128 * 128) / 512 = 32 blocs.

Question B3

Les registres adressables du contrôleur disque :
- BLOCK_DEVICE_BUFFER (0x00) – R/W
- BLOCK_DEVICE_COUNT (0x04) – R/W
- BLOCK_DEVICE_LBA (0x08) – WRONLY
- BLOCK_DEVICE_STATUS (0x10) – RDONLY
- BLOCK_DEVICE_IRQ_ENABLE (0x14) – R/W
- BLOCK_DEVICE_SIZE (0x18) – RDONLY
- BLOCK_DEVICE_BLOCK_SIZE (0x1C) – RDONLY

→Accès en **lecture** dans le registre *STATUS* dans les états : *READ/WRITE_(ERROR/SUCCESS)*, remet l'automate master dans l'état IDLE et acquitte les IRQ.
→ Accès en **écriture** dans les registres *BUFFER*, *COUNT*, *LBA*, *OP* est ignoré si le contrôleur disque est dans l'état *IDLE*.

Question B4

Les différentes valeurs de l'état interne du contrôleur de disque visible du logiciel :
- BLOCK_DEVICE_IDLE
- BLOCK_DEVICE_BUSY
- BLOCK_DEVICE_READ_SUCCESS
- BLOCK_DEVICE_WRITE_SUCCESS
- BLOCK_DEVICE_READ_ERROR
- BLOCK_DEVICE_WRITE_ERROR

C) *Architecture matérielle*

Question C1

L'utilisation du composant PibusBlockDevice impose l'augmentation de la valeur du *timeout* car lors d'une entrée/sortie, le composant PibusSegBcu doit attendre la fin de l'E/S. La valeur qu'il faut donner au *timeout* est la latence du contrôleur IOC : 1000 cycles.

Question C2

- Adresse de base du segment IOC : 0x92000000
- Taille du segment IOC : 32 octets

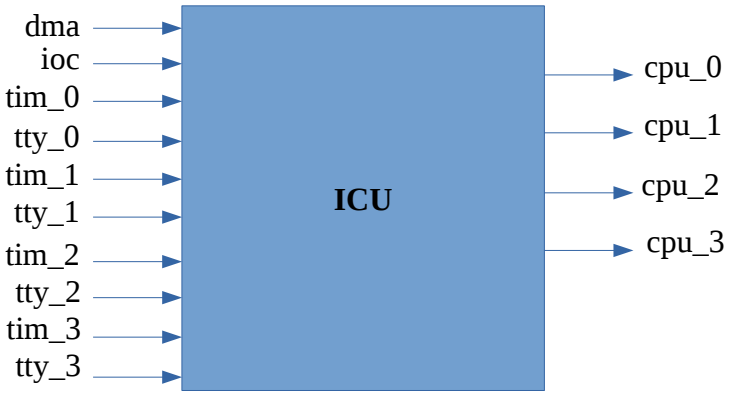
On est sur une architecture à 4 processeurs, la taille des segments :

- ICU 128 bytes
- TTY 64 bytes
- TIMER 64 bytes

Question C3

Dans cette architecture, on trouve :
- 6 composants maitres : DMA, IOC et 4 processeurs.
- 6 composants cibles : ICU, RAM, ROM, TTY, FBF, TIMER

Question C4



D) *Code de boot*

Question D1

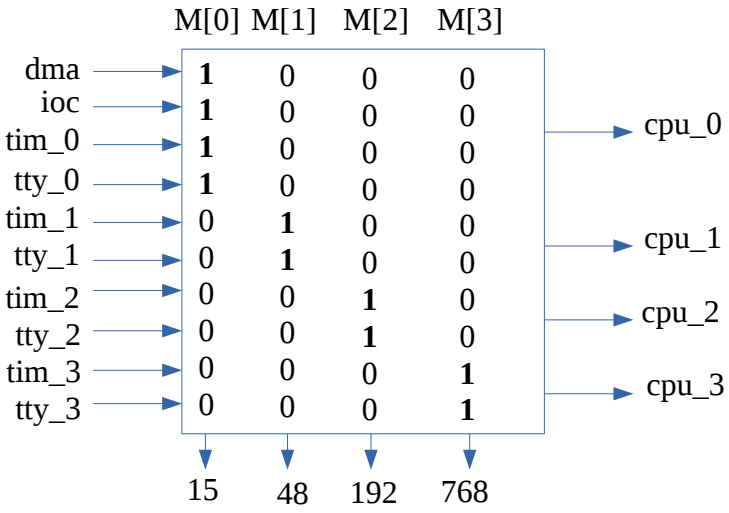
Chaque processeur doit initialiser son propre pointeur de pile parce que les segment sont allouées de façon statique vu qu'il n y a pas de MMU pour faire la traduction.

Question D2

Le système d'exploitation fait router les interruptions en utilisant l' « Interrupt masks » dans le PSW.

Question D3

Les valeurs à stocker dans les 4 registres de masque de l'ICU :



E) *Application logicielle de traitement de l'image*

Question E1

La fonction *ioc_read()* permet d'effectuer un transfert depuis le disque vers la mémoire.
Les arguments de l'appel système *ioc_read()* :
- lba Logical Block Address (first block index)
- base address of the memory buffer
- count number of blocks to be transfered

Cet appel système n'attend pas la fin du transfert pour rendre la main. Il est bloquant dans le cas ou le contrôleur disque est occupé (attente active (polling) sur le *_ioc_lock* associé au contrôleur).

Question E2

L'appel système *ioc_completed()* sert à indiquer la fin du transfert (I/O). Cette fonction ne possède pas d'arguments. En effet, cette fonction fait une attente active (polling) sur la variable *_ioc_done* dont l'état change à la réception d'une interruption disque indiquant la fin de l'entrée sortie.

Question E3

le dysfonctionnement observé est que l'image suivante ne s'affiche pas.
Ce problème est du au fait que les données du cache ne sont pas invalidées à chaque transfert d'une image.
La mise à 1 de la macro *NO_HARD_CC*(dans config.h), qui autorise à invalider les données du cache dans l'appel système *ioc_read()*, résout le problème.

L'activation du *SNOOP* permet aussi de résoudre ce problème. En effet, ce mécanisme sert à surveiller toutes les écritures passer sur le bus et donc d'invalider le cache de données.

Question E4

La condition qui fait sortir l'automate *SNOOP_FSM* de l'état *IDLE* est la détection d'une demande d'écriture externe. On cas de hit externe, la ligne de cache concernée est invalidée. Si par contre, plusieurs hits externes consécutifs ont été détectés, le cache est mis à jour.

Question E5

La détection de plusieurs hit externes consécutifs pose un problème car il y a plusieurs types de hit-externe et dans chacun des cas, l'automate *SNOOP_FSM* doit procéder différemment.

Les 3 types de hit externes sont :
a - the external write matches a locally cached line.
b - the external write matches a requested cache line.**Question**
c - the external write matches a pending llsc address.

Dans le cas (a) ou (b), le *SNOOP_FSM* demande au *DCACHE_FSM* d'invalider la ligne de cache concernée en utilisant :
→ r_snoop_dcache_inval_req (bascule)
→ r_snoop_dcache_inval_set
→ r_snoop_dcache_inval_way
Si entre temps, un autre hit externe est détecté, le *SNOOP_FSM* rentre dans un état de panic ou il va demander (avant la fin de la première opération) au *DCACHE_FSM* de mettre à jour le cache de données en utilisant :
→ r_snoop_dcache_flush_req (bascule)
Dans le cas (c), le *SNOOP_FSM* demande au *DCACHE_FSM* d'invalider la réservation LLSC via :
→ r_snoop_llsc_inval_req (bascule)

Donc, vu les différents cas, le problème est résolu en **mettant à jour le cache**.

Question E6

Les durées d'exécution des trois étapes (chargement, filtrage, affichage) des 4 premières images :

	Load	Filtrage	Print
Image 01	41816	439198	120433
Image 02	40539	439382	120138
Image 03	40593	439382	120138
Image 04	40593	439382	120138

F) *Exécution sur architecture multi-processeurs*

Question F1

La phase qui va être paralléliser est uniquement l'étape de filtrage. En effet, les appels systèmes *ioc_read()* et *fb_sync_write()* sont bloquants, donc il y a qu'un seul processeur qui avance à un instant 't'.

Question F2

Le mécanisme générale qui permet de séquentialité les 4 transferts demandés par les 4 processeurs c'est de laisser à un seul processeur d'utiliser le contrôleur *IOC* pendant qu'on que les autres restent bloqués sur un verrou.

Question F3

La fonction *_ioc_get_lock()* utilise les deux instructions assembleurs offertes par le processeur MIPS (Linked Load – LL, Store Conditionnel – SC) pour réaliser la prise d'un verrou (*_ioc_lock*) de façon atomique.

Question F4

La fonction système qui libère le rôleur IOC ne pouvrant effectuer qu'un seul transfert à la verrou est *_ioc_completed()*. Elle se contente de remettre le *_ioc_lock* à zéro.
Il n'est pas nécessaire d'utiliser une instruction particulaire pour libérer le verrou car le processus est le seul à pouvoir le détenir.

G)*Réalisation matérielle du LL/SC*

Question G1

L'enregistrement de la prise de réservation se fait du coté du processeur et non pas du coté de la mémoire car la taille de la table de réservation associative dans le contrôleur mémoire pose des problèmes.

Question G2

Le fait que l'on dispose d'un bus et donc d'un mécanisme de *SNOOP*, la **prise de réservation** et **l'enregistrement de l'adresse** et **la surveillance** peut être réaliser par les contrôleurs de cache Maitres (un seul registre par cache L1). Le processeur P0 est informé de l'écriture effectuée par P1 par l'invalidation, effectuée par le *SNOOP*, de la réservation avec la bascule « r_snoop_llsc_inval_req ».

Question G3

Âpres avoir désactiver le *SNOOP*, on observe un dysfonctionnement sur l'affichage de la première image : un processeur parmi les 4 échoue sont affichage (affiche une portion noir).

Question G4

Dans ce TP, le mécanisme de *SNOOP* met en évidence l'invalidation du cache de données et l'invalidation de la réservation prise par l'instruction « Linked Load ».