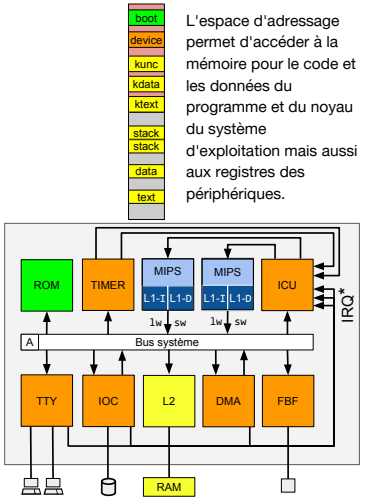
ALMO

Architecture Multi-Cœurs (multicores)

Architecture de la plateforme multi-cœurs

La plateforme contient :

- MIPS32 : 2 coeurs avec ses caches L1
- ROM : mémoire de code de boot
- L2 : un cache L2
- TIMER : un compteur de temps
- TTY : un contrôleur de terminaux
- IOC : un contrôleur de disque
  In Out Controler
- DMA : un opérateur de copie
  Direct Memory Access
- FB : un contrôleur vidéo
  Frame Buffer
- ICU : un concentrateur d'IRQ



L'espace d'adressage permet d'accéder à la mémoire pour le code et les données du programme et du noyau du système d'exploitation mais aussi aux registres des périphériques.

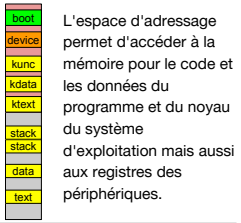
Architecture de la plateforme multi-cœurs

Dans cette plateforme, il y a :

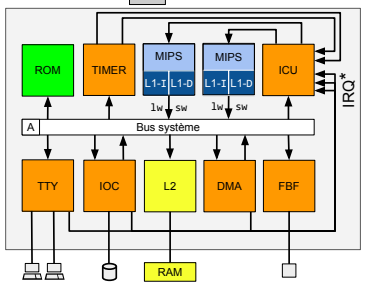
- autant de TTY qu'il y a de tâches et de cœurs.
- autant de DMA qu'il y a de cœurs
- autant de TIMER qu'il y a de cœurs
- mais un seul IOC et un seul FB
- une ICU capable de router les signaux d'interruption vers les cœurs

Il y a au au moins deux stacks
dans l'espace d'adressage :

- Chaque cœur utilise sa propre pile.



L'espace d'adressage permet d'accéder à la mémoire pour le code et les données du programme et du noyau du système d'exploitation mais aussi aux registres des périphériques.



Plan

Le but de ce cours est de comprendre les conséquences sur le matériel et sur le logiciel, puis d'étudier le gain sur les performances.

- Composants multi-canaux de la plateforme
- Routage des signaux d'interruption vers les cœurs
- Rôle et fonctionnement de l'arbitre du bus
- Gestion de la cohérence des caches
- Conséquences sur le noyau
- Gain en performances

Composants de plateforme multi-cœurs

- ROM : Un seul code de boot pour tous les cœurs
- L2 : un cache L2 unique qui permet de gérer les lignes de l'espace d'adressage partagé par les cœurs
- TIMER : autant de timer que de cœurs
  chaque timer permet de produire un tick d'horloge.
- TTY : Autant de terminaux qu'il y a de cœurs et de tâche par cœur.
- IOC : un contrôleur de disque unique et l'IRQ du disque est géré par cœur unique, mais les deux peuvent écrire des requêtes.
- DMA : un opérateur de copie par cœur.
- FB : un contrôleur vidéo unique pour les tous les cœurs
- ICU : l'ICU a autant de MASK que de cœur afin de router les signaux d'interruption vers un cœur.

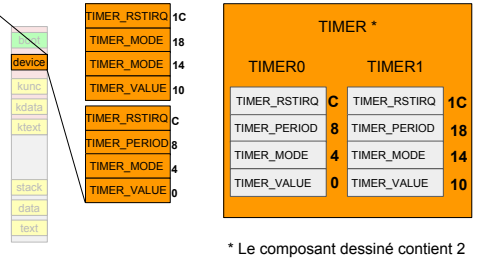
TIMER

Il faut un TIMER par cœur puisque chaque cœur doit recevoir une requête d'interruption et interrompre la tâche courante pour que l'ordonnanceur choisisse une nouvelle tâche.

- TIMER_VALUE (lecture/écriture) +1 à chaque cycle
- TIMER_MODE (écriture seule) configure le mode de fonctionnement
  - Bit 0: 1 = timer en marche (décompte) ; 0 = timer arrêté
  - Bit 1: 0 = pas d'IRQ quand le compteur atteint 0
- TIMER_PERIOD (écriture seule) période entre 2 IRQ
- TIMER_RESETIRQ (écriture seule) écrire à cette adresse acquitte l'IRQ

Les TIMERs sont dans le même composant mais sont totalement indépendants.

Il y a autant d'IRQ que de TIMER.



* Le composant dessiné contient 2 timers, le nombre est configurable.

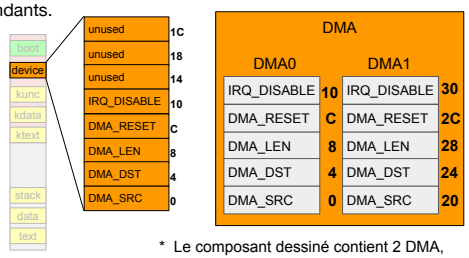
DMA

Notre plateforme suppose un canal DMA par cœur.

- DMA_IRQ_DISABLE (lecture/écriture) masquage de la ligne IRQ
- DMA_RESET (écriture seule) acquittement de la ligne IRQ
- DMA_LEN (écriture/lecture) taille en octets à déplacer
- DMA_DST (écriture seule) adresse de destination
- DMA_SRC (écriture seule) adresse source

Les canaux DMA sont indépendants.

Chaque canal DMA a son IRQ vers un coeur (c.f. ICU) mais ils sont dans le même composant ils se partagent donc la bande passante du port de sortie.



* Le composant dessiné contient 2 DMA, dans la plateforme, on en met un par MIPS

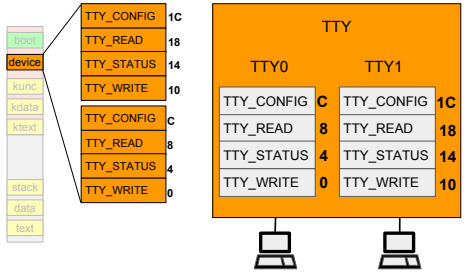
TTY

Notre plateforme suppose qu'il y a autant de terminaux que de cœurs et de tâches par cœur. Les fonctions de l'API la libc (ici: stdio) ne demande pas de préciser le terminal. C'est le GIET qui détermine le bon terminal en fonction de la tâche qui demande.

- TTY_WRITE (écriture seule) port de sortie vers le terminal
- TTY_STATUS (lecture seule) informe qu'on peut lire ou envoyer un caractère
- TTY_READ (lecture seule) port d'entrée depuis le clavier
- TTY_CONFIG (écriture seule) configuration du protocole d'envoi (non utilisé)

Tous les terminaux sont indépendants.

Chaque terminal a sa/ses ligne/s d'interruption.



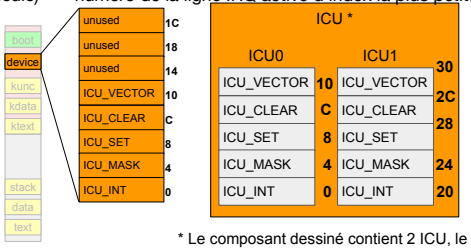
ICU

Dans le cas d'une architecture multicœurs, il permet de router les IRQ vers n'importe quel core.
Les IRQs de tous les périphériques entre dans l'ICU et chaque canal est configuré pour laisser passer une sélection d'IRQ pour chaque cœur

- ICU_INT (lecture seule) état des lignes IRQ (commun à tou les canaux)
- ICU_MASK (lecture seule) masques des lignes IRQ
- ICU_CLEAR (écriture seule) commande de mise à 0 des masques d'IRQ
- ICU_SET (écriture seule) commande de mise à 1 des masques d'IRQ
- ICU_VECTOR (lecture seule) numéro de la ligne IRQ active d'index la plus petite

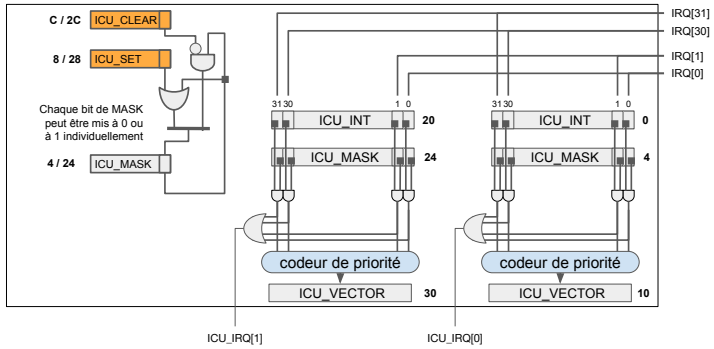
Pour le GIET
C'est dans le code RESET que l'on choisit quelles IRQs vont quel cœur.

Il y a autant de sortie IRQ que de cœurs.



* Le composant dessiné contient 2 ICU, le nombre est configurable.

Architecture de la plateforme / ICU

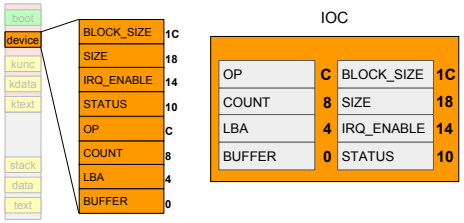


Périphérique IOC : In Out Controler

Le composant IOC est de la plateforme n'a qu'un seul canal. Il n'y a qu'un seul disque et le contrôleur ne peut gérer qu'une seule requête à la fois.

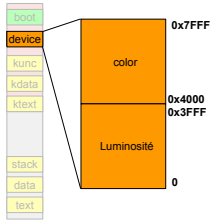
- BLOCK_SIZE (lecture seule) taille d'un bloc en octets
- SIZE (lecture seule) taille du disque en bloc
- IRQ_ENABLE (lecture/écriture) masquage de la ligne IRQ
- STATUS (lecture) état de l'IOC en fin d'opération (acquitte IRQ)
- DMA_OP (écriture seule) acquittement de la ligne IRQ
- DMA_COUNT (écriture/lecture) taille en octets à déplacer
- DMA_LBA (écriture seule) adresse de destination (en bloc)
- DMA_BUFFER (écriture seule) adresse source (adr. octets alignée sur un bloc)

Il n'y a qu'une seule IRQ sortant de l'IOC



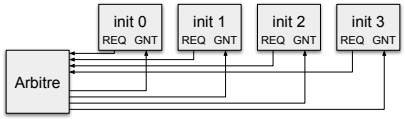
Périphérique FBF : Frame Buffer

Le composant Frame Buffer est un périphérique cible.
Il n'a de registre adressage. C'est une mémoire accessible en écriture et en lecture.
Les valeurs que l'on écrit sont envoyé vers l'écran.

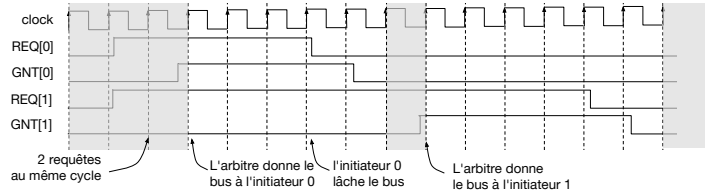


Arbitre du BUS

- Dès lors qu'il y a plusieurs initiateurs, il faut un arbitre pour décider quel initiateur à le droit d'utiliser le bus.
- Chaque initiateur communique sa demande du BUS par un signal REQ (request) et reçoit de l'arbitre un signal d'autorisation par un signal GNT (grant).



- Un signal REQ[i] est maintenu jusqu'à la fin de la transaction de l'initiateur [i]
- Un signal GNT[i] actif informe que le bus est utilisable par par l'initiateur [i]



Politique d'arbitrage

Lorsque plusieurs initiateurs demandent le bus simultanément, l'arbitre choisit l'ordre selon une politique d'arbitrage :

Priorité fixe

- Consiste à donner une priorité fixe à chaque initiateur, par exemple core0 > core1 > DMA > IOC
  ⇒ Risque de famine si core0 utilise très souvent le bus

Priorité tournante (ou round-robin)

- Consiste à donner la priorité à l'initiateur dont l'usage du bus est le plus ancien. Donc, si un initiateur lâche le bus, il devient le moins prioritaire.

Dans un système généraliste, la priorité fixe est un problème à cause des risques de famine. L'arbitre de notre plateforme utilise une priorité tournante.

Bande passante du bus

Définition

- En informatique, la bande passante d'un interconnect correspond au volume de données que l'on peut échanger par unité de temps.
- Elle est mesurée, par exemple, en octets / seconde ou en octets / cycle.
- Sur notre plateforme, la bande passante est de 4 octets / cycles.
  Pour l'avoir en octets / seconde, il faut connaître la durée d'un cycle.

Usage

- Chaque initiateur utilise une partie de la bande passante en réalisant des transactions de lectures ou d'écritures.
- La quantité de bande passante utilisée par un initiateur est mesurée en pourcentage, elle dépend de la fréquence des transactions demandées et de la durée de ces transactions.
- On ne peut pas utiliser plus que 100% de la bande passante, il y a donc une limite au nombre d'initiateurs sur un bus.

Evaluation de la bande passante nécessaire par cœur

Fréquence des transactions (F_i)

- Il faut commencer par déterminer les transactions possibles :
  - Lecture d'instruction
  - Lecture de données cachées
  - Lecture de données non cachées
  - Ecriture de données
- Pour chaque transaction, il faut connaître sa fréquence (par cycle).
- Par exemple, on sait que le cœur exécute une instruction d'écriture (sw) toutes les N instructions. Le cache est write-through, toutes les écritures sont envoyées sur le bus. On connaît une estimation du CPI ⇒ La fréquence d'écriture par cycle est 1/(N*CPI)
- Pour simplifier on peut déterminer le nombre de transactions demandées sur une période de 100 cycles et diviser par 100 pour avoir le nombre de transaction par cycle, c'est à dire sa fréquence.

Durée de transaction (T_i)

- Correspond à la durée d'usage du bus pour réaliser une transaction (en cycles)

La Bande passante nécessaire pour une transaction = $(F_i * T_i)$ (c'est un %)

Pour le cœur, il faut sommer la part de chaque transaction ⇒ c.f. TME.

Cycles de gel dûs au BUS

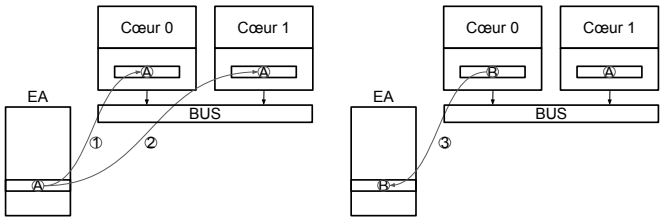
- Dès lors qu'il y a plusieurs initiateurs, le bus n'est pas toujours disponible.
- Si un cache veut faire une lecture de ligne et qu'il ne peut utiliser le bus parce que celui-ci est déjà utilisé, il va devoir attendre et donc geler le cœur pendant cette attente (cela s'ajoute au gel nécessaire à la lecture de la ligne).

  ⇒ Comment estimer le gel du cœur dû à l'attente du BUS
- Pour les écritures, c'est 0 parce que les écritures sont postées dans un tampon d'écriture, si on suppose que celui-ci n'est jamais plein.
- Pour les lectures
  - Dans le meilleur des cas c'est 0
  - Dans le pire des cas c'est (C-1) * T_{max}
    - avec C le nombre de cœurs
    - et T_{max} la durée de la plus longue transaction.

Cycles de gel dûs au BUS

- Pour être plus réaliste, il faut estimer la durée moyenne d'une transaction T_{moy}
  - C'est la durée des transactions sur une période de temps divisée par le nombre de transactions pendant la même période.
  - Si on raisonne sur 100 cycles, c'est le nombre de cycles utilisés pour les transactions divisé par le nombre de transactions en 100 cycles.
  - Si on raisonne sur 1 cycle : c'est la bande passante utilisés par les transactions divisé par la somme des fréquences de transactions
- Une fois estimé la durée moyenne d'une transaction, pour estimer le temps d'attente du bus et donc la durée du gel, il faut connaître la politique de l'arbitre et le nombre de cœurs.
- Si on a C cœurs, en fonction de sa priorité on peut :
  - ne pas attendre ou attendre T_{moy} ou attendre $2*T_{moy}$, etc. ou attendre $(C-1)*T_{moy}$
  - Le temps moyen d'attente $\sum (a*T_{moy}) / C$ pour a = 0 à C-1

Cohérence de cache



Ce schéma illustre le problème de cohérence

1. Le Cœur 0 lit une ligne contenant le mot A
2. Le Cœur 1 lit la même ligne contenant le mot A
3. Le Coeur 0 écrit B dans la ligne, son cache se met à jour et il écrit en mémoire.

⇒ Le cache du cœur 1 n'est plus cohérent vis-à-vis de la mémoire

Solutions hardware : ne pas utiliser le cache ou snooping réalisé par les caches
Solution software : flush des lignes de caches

La semaine prochaine

- La plateforme a plusieurs cœurs avec plusieurs tâches par cœur.
- Toutes les tâches se partagent le même espace d'adressage.
- Chaque tâche a besoin de trois segments : text, data et stack.
- Il faut veiller à ce que chaque tâche utilise ses propres segments

⇒ il faut produire le code binaire en fonction de l'usage de l'EA.

Question
Comment faire pour que chaque tâche dispose de son propre EA.

Réponse
Définir un mécanisme placé entre un cœur et ses caches générant un espace d'adressage virtuel