ALMO

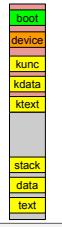
GIET

Périphériques DMA (initiateurs)

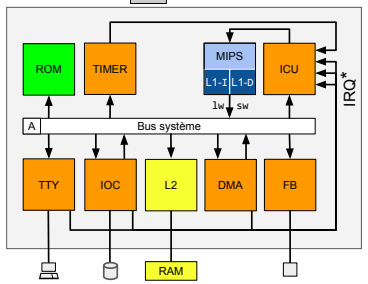
Architecture de la plateforme

La plateforme contient :

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



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.



Types de périphériques

On distingue deux types de périphériques :

Périphérique cible : TTY, ICU, TIMER, FB

- Un périphérique cible reçoit des requêtes de lecture ou d'écriture dans ses registres pour réaliser des opérations. Ses registres de contrôle sont mappés dans l'espace d'adressage physique pour être accessible par un programme.
- Un périphérique cible peut lever des IRQ pour informer d'un événement (nouvelle donnée ou fin de traitement d'une opération demandée)

Périphérique initiateur : DMA, IOC

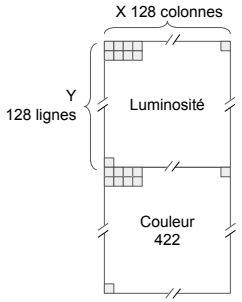
- Un périphérique initiateur reçoit également des requêtes de lecture ou d'écriture dans ses registres pour réaliser des opérations. Ses registres de contrôle sont aussi mappés dans l'espace d'adressage physique pour être accessible par un programme.
- Un périphérique initiateur peut aussi lever des IRQ pour informer d'un événement (nouvelle donnée ou fin de traitement d'une opération demandée).
- Ce qui le distingue, c'est qu'un **périphérique initiateur peut faire des requêtes** de lecture et d'écriture dans l'espace d'adressage.

Périphérique FB : Frame Buffer

C'est un contrôleur vidéo, ici c'est une simple mémoire représentant l'image codé YUV422

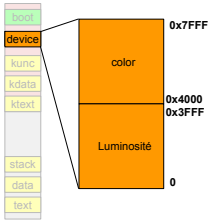
- Sur la plateforme l'image fait 128 x 128 pixels.
- Chaque pixel occupe deux octets : 1 pour la luminosité, 1 pour la couleur
- L'image occupe donc 128 x 128 x 2 = 32KiB = 16 + 16
- Le premier octet contient le niveau de gris du pixel (0,0) en haut à gauche, le second est le pixel (1,0), etc jusqu'au pixel (127,0) puis l'octet 128 contient le pixel (0,1), etc.
- Le tableau des couleurs est situé juste après avec les pixels rangés dans le même ordre

Luminosité	0 = noir 255 = blanc
Couleur	4 bits rouge 2 bits vert 2 bits bleu



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.

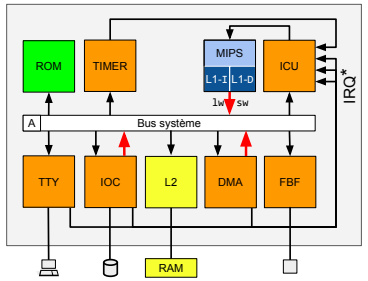


Périphériques initiateurs

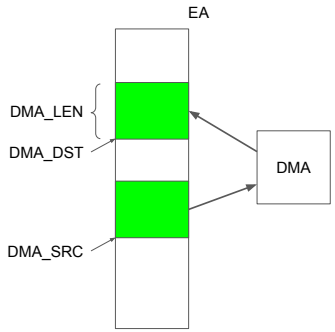
IOC (In Out Controler) : contrôleur de disque
- Le processeur écrit une commande de lecture ou d'écriture du disque.
- L'IOC réalise la commande en faisant les accès au disque et à la mémoire.
- Après la terminaison de l'opération de transfert, il lève une IRQ (interruption).

DMA (Direct Memory Access) : Opérateur de copie mémoire
- Le processeur écrit une commande de déplacement : src, dst, size
- Le DMA réalise l'opération en accédant directement à la mémoire
- Après la terminaison de l'opération de transfert, il lève une IRQ.

L'arbitre va gérer l'accès au bus
pour les 3 initiateurs : MIPS, DMA et IOC



Périphérique DMA : Direct Memory Access



Le composant reçoit des ordres de copie

Si c'est l'utilisateur qui fait la demande, il passe par un appel système, sa demande est vérifiée afin de lui interdire d'écrire dans la segment d'adresse réservé au système
(> 0x80000000)

Le système doit également géré le fait que le *canal* DMA est libre. En effet, celui peut être déjà utilisé par un autre programme. Si le canal DMA est déjà utilisé, le nouveau demandeur doit être mise en attente, jusqu'à la terminaison de la copie courante.

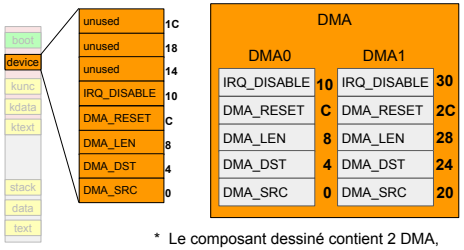
Périphérique DMA : Direct Memory Access

Le DMA réalise un copie de mémoire à partir de l'adresse DMA_SRC vers l'adresse DMA_DST de DMA_LEN octets.
Dans l'ordre, on commence par écrire les adresses SRC, DST et IRQ_DISABLE (si besoin), **puis on écrit LEN, ce qui provoque le démarrage de la copie par le DMA**

• 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

A la fin de l'opération, le DMA lève un interruption et LEN contient le nombre d'octet non écrits.

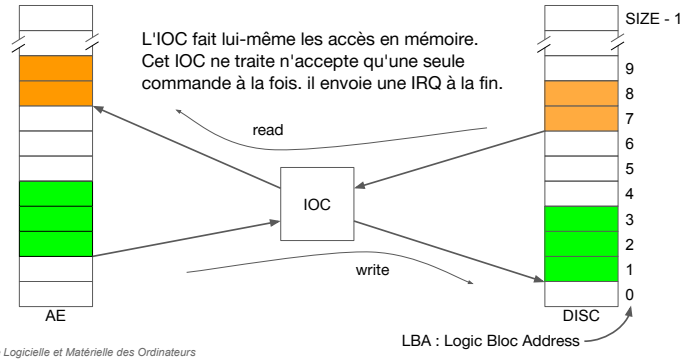
S'il est différent de 0, c'est qu'il y a une erreur.



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

Périphérique IOC : In Out Controler

- Le contrôleur de disque permet de lire (read) ou d'écrire (write) le disque.
- Les échanges se font par blocs. La taille d'un Bloc est une caractéristique intrinsèque du disque, c'est typiquement 512 octets (B@LOCK_SIZE)
- Les opérations sont plus ou moins longues en fonction de la nature du disque. Sur un disque mécanique, il faut déplacer une tête de lecture avant de lire ou d'écrire, cela peut être long. Sur un disque fash, il n'y a pas de latence.



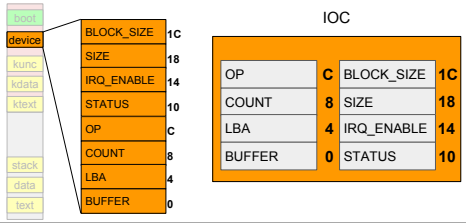
Périphérique IOC : In Out Controler

Le composant IOC est aussi appelé Bloc Device parce qu'il déplace des blocs du disque.
Dans la table ci-dessous, le nom des registres est préfixé par BLOC_DEVICE_ (cf. p. 37)
Dans l'ordre, on commence par écrire : BUFFER, LBA, COUNT et IRQ_ENABLE (si besoin),
puis on écrit OP, ce qui provoque le démarrage de l'IOC

• 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)
• OP	(écriture seule)	sens de la transaction (read du disque ou write du disque) l'écriture dans ce registre démarre la transaction
• COUNT	(écriture/lecture)	taille en blocs à déplacer
• LBA	(écriture seule)	adresse de destination (en bloc)
• BUFFER	(écriture seule)	adresse source (adr. octets alignée sur un bloc)

A la fin de l'opération,
l'IOC lève un interruption et
COUNT contient le nombre
de BLOCK non écrits.

S'il est différents de 0,
c'est qu'il y a une erreur.



Périphérique IOC : In Out Controler

p. 37 de doc giet se trouve les opérations possibles et les valeurs de status.

```
enum IOC_operations {
    BLOCK_DEVICE_NOOP ,              // valeur par défaut
    BLOCK_DEVICE_READ ,              // demande de lecture du disque
    BLOCK_DEVICE_WRITE ,             // demande d'écriture sur le disque
};
enum IOC_status{
    BLOCK_DEVICE_IDLE ,              // valeur par défaut, si l'IOC ne fait rien
    BLOCK_DEVICE_BUSY ,              // IOC est déjà en train de faire une opération
    BLOCK_DEVICE_READ_SUCCESS ,      // IOC a terminé avec succès et COUNT == 0
    BLOCK_DEVICE_WRITE_SUCCESS ,     // IOC a terminé avec succès et COUNT == 0
    BLOCK_DEVICE_READ_ERROR ,        // IOC n'a pas pu terminer, COUNT != 0
    BLOCK_DEVICE_WRITE_ERROR ,       // IOC n'a pas pu terminer, COUNT != 0
    BLOCK_DEVICE_ERROR ,             // IOC a une erreur fatale (ex. disque perdu)
};
```


Cohérence de cache : le problème

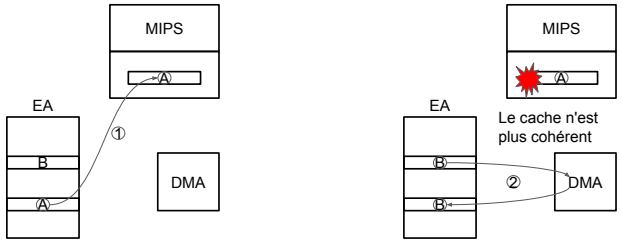
- Les caches L1 contiennent des copies de lignes de caches de la mémoire.
- On dit qu'un cache est cohérent s'il contient des copies à jour, c'est-à-dire avec les dernières modifications de la ligne.
- Si la plateforme ne contient d'un seul cœur et qu'il est le seul à faire des accès dans la mémoire alors les caches (I+D) sont toujours à jour.
  - Le cœur lit des lignes, il place les copies dans ses cases.
  - Si le cœur écrit un mot, il l'écrit en mémoire (write-through) et si la ligne est présente dans le cache, celle-ci est mise à jour.
- Si le cœur n'est pas le seul à faire des écritures dans la mémoire, il y a un problème de cohérence.
  - Le cache peut avoir la copie d'une ligne dans l'une de ses cases.
  - Si un autre composant modifie cette ligne en mémoire, **alors la copie de la ligne présente dans le cache n'est plus à jour**, et si le cœur lit cette ligne, il y a HIT mais la donnée rendue n'est pas bonne.

Cohérence de cache : illustration

Ce schéma illustre le problème de cohérence

1. Le MIPS lit une ligne contenant le mot A
2. le DMA reçoit un ordre de copie ayant pour destination la ligne de A.
   ⇒ La ligne contenant A contient désormais B,
   mais le cache du MIPS contient toujours A

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



Cohérence de cache : solutions

Pour résoudre le problème de cohérence, il existe 2 types de solutions

- Solution hardware : snooping (espionnage)
  - Si les initiateurs accèdent à l'espace d'adressage par un bus commun ils peuvent espionner (to snoop) les transactions et en particulier les écritures.
  - Si un cache voit une écriture dans une ligne dont il possède une copie, il peut soit la mettre à jour, soit simplement invalider la copie

- Solution Software : cache flush
  - C'est l'OS qui envoie les commandes vers les autres initiateur.
  - Quand il commande un initiateur (DMA ou IOC) il sait exactement quelles sont les lignes qui vont être écrites.
  - Il peut demander au cache d'invalider ces lignes par avance.
  - Ainsi quand le cœur accèdera à la ligne, il y aura MISS et la dernière version sera lue.
  - Il peut aussi invalider tout le cache...

Gestion du partage de ressources

Quand un composant est en un seul exemplaire, qu'il ne peut exécuter qu'une seule requête à la fois et qu'il peut potentiellement recevoir plusieurs commandes venant de plusieurs tâche, il y a un problème de partage.

Le système doit offrir un moyen de bloquer toute nouvelle commande.
⇒ le système va utiliser un verrou à attente active (spinlock)

un spinlock est représenté :

- par une case mémoire **lock** pouvant prendre 2 états : 0 == libre, 1 == occupé
- par 2 "fonctions" d'accès : prendre et relâcher
  - prendre : réalise de manière atomique : while (lock==1); lock = 1
  - relâcher : réalise : lock = 0
- attention le lock ne doit pas être caché, ni par le cache, ni par gcc

Fonction de prise de lock

```
static inline void _ioc_get_lock()
{
    register unsigned int delay = (_proctime() & 0xF) << 4;
    register unsigned int * plock = (unsigned int *) &_ioc_lock;

    asm volatile (
            "_ioc_llsc:␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
            "ll␣␣␣$2,␣␣␣␣0(%0)␣␣␣␣␣␣␣\n" /* $2 <= _ioc_lock current value */
            "bnez␣$2,␣␣␣␣_ioc_delay␣\n" /* delay if _ioc_lock already taken */
            "li␣␣␣$3,␣␣␣␣1␣␣␣␣␣␣␣␣␣\n" /* $3 <= argument for sc */
            "sc␣␣␣$3,␣␣␣␣0(%0)␣␣␣␣␣␣␣\n" /* try to set _ioc_lock */
            "bnez␣$3,␣␣␣␣_ioc_ok␣␣␣␣\n" /* exit if atomic */
            "_ioc_delay:␣␣␣␣␣␣␣␣␣␣␣␣\n"
            "move␣$4,␣␣␣␣%1␣␣␣␣␣␣␣␣␣\n" /* $4 <= delay */
            "_ioc_loop:␣␣␣␣␣␣␣␣␣␣␣␣\n"
            "addi␣$4,␣␣␣␣$4,␣␣␣␣-1␣␣\n" /* $4 <= $4 - 1 */
            "bnez␣$4,␣␣␣␣_ioc_loop␣␣\n" /* test end delay */
            "j␣␣␣␣␣␣␣␣␣␣␣_ioc_llsc␣␣\n" /* retry */
            "_ioc_ok:␣␣␣␣␣␣␣␣␣␣␣␣␣\n"
            :
            :"r"(plock), "r"(delay)
            :"$2", "$3", "$4");
}
```

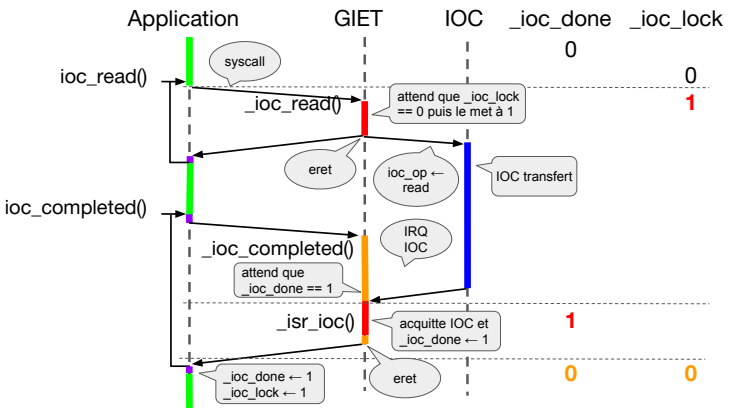
Synchronisation entre l'application et le périphérique

Le périphérique IOC met du temps à réaliser les commandes.
Il y a deux attitudes possibles, attendre ou faire autre chose en parallèle.
Si on choisit de faire autre chose en parallèle, il faut un moyen de savoir quand la commande précédente est terminée.

Le système utilise une variable globale indiquant de l'opération en cours est terminée : _ioc_done

- cette variable est à 0 au démarrage
- l'application prend l'IOC et fait sa commande et reprend ses travaux.
- l'IOC commence à faire son transfert
- lorsque l'IOC termine, elle lève une IRQ
- qui provoque l'exécution _isr_ioc() dont le travail sera
  - de lire le résultat status
  - d'acquitter l'interruption
  - mettre à 1 _ioc_done
- Quand le programme le veut il peut se mettre en attente de _ioc_done à 1 dans une attente active.

Synchronisation : Application - Périphérique



Lecture du disque : appel système

```
/*
 * ioc_read()
 *
 * Transfer data from a file on the block_device to a memory buffer.
 * - lba    : Logical Block Address (first block index)
 * - buffer : base address of the memory buffer
 * - count  : number of blocks to be transfered
 *
 * - Returns 0 if success, > 0 if error (e.g. memory buffer not in user space).
 */
unsigned int ioc_read(unsigned int lba, void * buffer, unsigned int count)
{
    return sys_call(SYSCALL_IOC_READ,
            lba,
            (unsigned int) buffer,
            count,
            0);
}
```


Lecture du disque : commande de l'IOC

```
/*
 * _ioc_read()
 *
 * Transfer data from a file on the block device to a memory buffer. The destination
 * memory buffer must be in user address space.
 * - lba    : first block index on the disk.
 * - buffer : base address of the memory buffer.
 * - count  : number of blocks to be transfered.
 *
 * - Returns 0 if success, > 0 if error.
 *
 * Note: all cache lines corresponding to the the target buffer are invalidated
 * for cache coherence.
 */
unsigned int _ioc_read(unsigned int lba, void * buffer, unsigned int count)
{
    volatile unsigned int * ioc_address;

    ioc_address = (unsigned int *) &seg_ioc_base;

    /* parameters checking */
    /* buffer must be in user space */
    unsigned int block_size = ioc_address[BLOCK_DEVICE_BLOCK_SIZE];

    if (((unsigned int) buffer >= 0x80000000)
            || (((unsigned int) buffer + block_size * count) >= 0x80000000))
        return 1;

    /* get the lock on ioc device */
    _ioc_get_lock();

    /* invalidation of data cache */
    if (NO_HARD_CC) _dcache_buf_invalidate(buffer, block_size * count);

    /* block_device configuration for the read transfer */
    ioc_address[BLOCK_DEVICE_BUFFER] = (unsigned int) buffer;
    ioc_address[BLOCK_DEVICE_COUNT] = count;
    ioc_address[BLOCK_DEVICE_LBA] = lba;
    ioc_address[BLOCK_DEVICE_IRQ_ENABLE] = 1;
    ioc_address[BLOCK_DEVICE_OP] = BLOCK_DEVICE_READ;


    return 0;
}
```

Lecture du disque : au moment de l'IRQ

```
/*
 * _isr_ioc
 *
 * There is only one IOC controler shared by all tasks. It acknowledge the IRQ
 * using the ioc base address, save the status, and set the _ioc_done variable
 * to signal completion.
 */
void _isr_ioc()
{
    volatile unsigned int * ioc_address;

    ioc_address = (unsigned int *) &seg_ioc_base;

    _ioc_status = ioc_address[BLOCK_DEVICE_STATUS]; /* save status & reset IRQ */
    _ioc_done   = 1;                                /* signals completion */
}
```

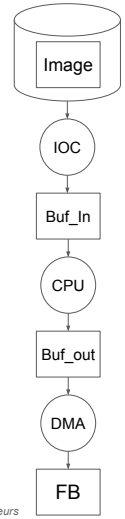

Lecture du disque : attente par l'application

```
/*
 * _ioc_completed()
 *
 * This function checks completion of an I/O transfer and reports errors. As it
 * is a blocking call, the processor is stalled until the next interrupt.
 *
 * - Returns 0 if success, > 0 if error.
 */
unsigned int _ioc_completed()
{
    unsigned int ret;

    /* busy waiting */
    while (_ioc_done == 0)
        asm volatile("nop");

    /* test IOC status */
    if ((_ioc_status != BLOCK_DEVICE_READ_SUCCESS)
          && (_ioc_status != BLOCK_DEVICE_WRITE_SUCCESS))
        ret = 1;        /* error */
    else
        ret = 0;        /* success */

    /* reset synchronization variables */
    _ioc_done = 0;
    _ioc_lock = 0;

    return ret;
}
```


Fonctionnement séquentiel

On veut afficher une image après l'avoir transformée.

Les trois composants initiateurs vont travailler l'un après l'autre



Le disque contient l'image

L'IOC fait le transfert en mémoire

Buffer en mémoire

Transformation

Buffer en mémoire

Le DMA transfert l'image dans le FB

Fonctionnement en pipeline

En utilisant 2 jeux de buffer en alternance on peut faire travailler en parallèle les trois composants : IOC, CPU et DMA.