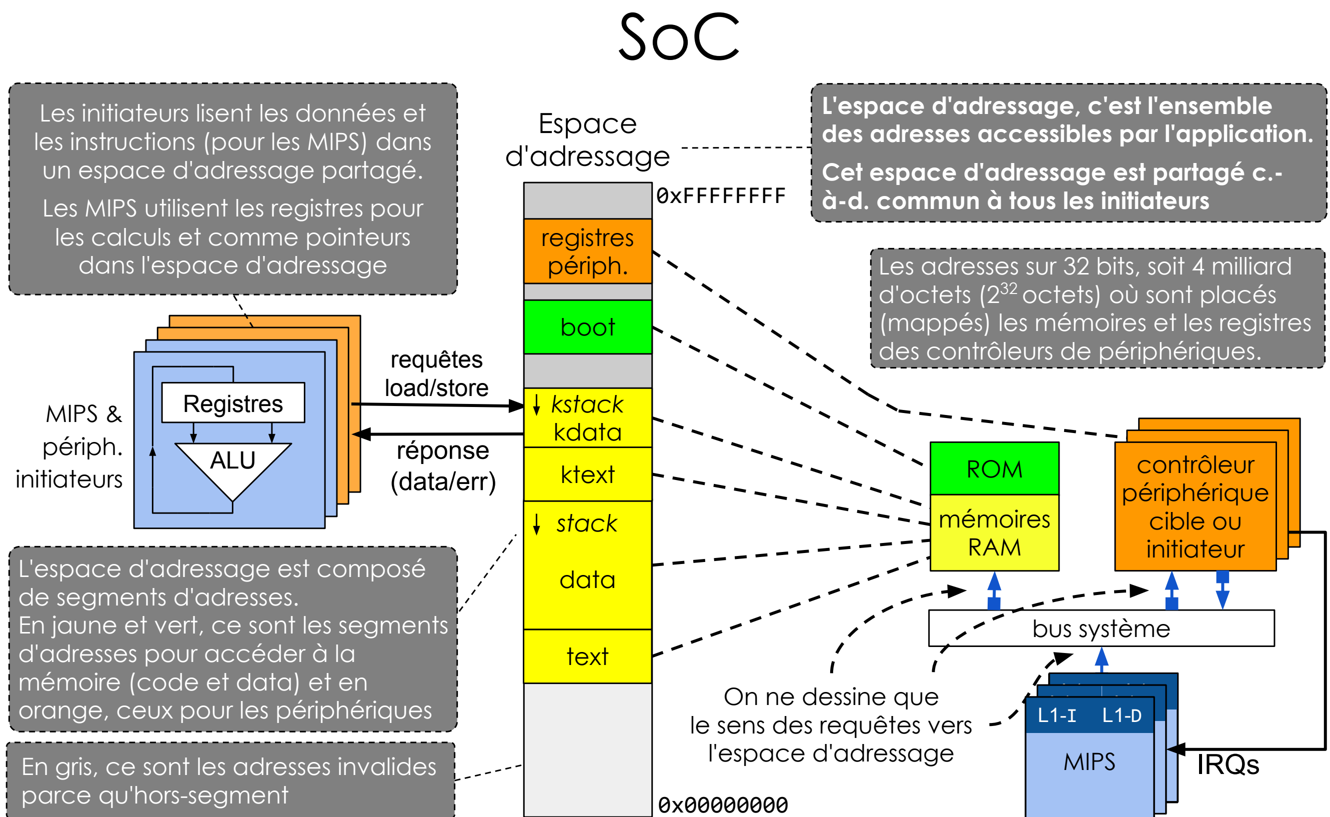
Interruptions

LU3IN031 Architecture des ordinateurs - 2
Matériel et Logiciel
B2

franck.wajsburt@lip6.fr

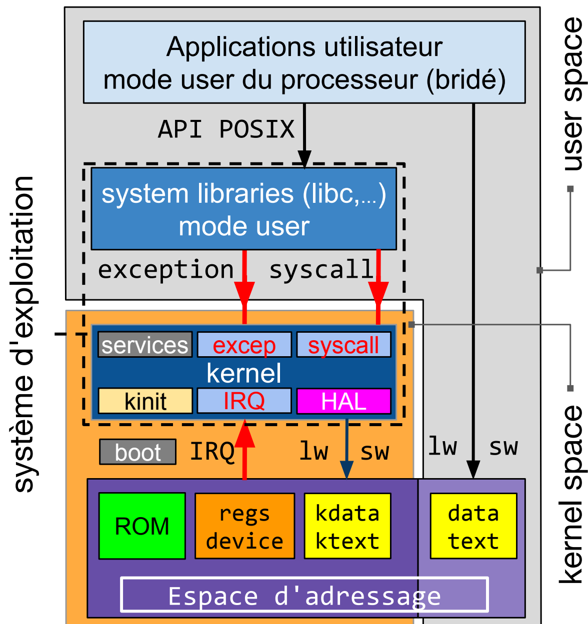
V3

SoC



Système d'exploitation

Le noyau contient les gestionnaires d'appels système, d'interruptions et d'exceptions



Les bibliothèques système offrent une API de service pour les applications qui font des syscalls au noyau, parfois des exceptions

1. Le **gestionnaire syscall** offre les services des sys. lib. p. ex :
   - les commandes de périphériques
   - l'allocation de mémoire
   - le lancement et l'arrêt d'application
2. Le **gestionnaire d'exception** gère les erreurs d'exécution du programme (le processeur ne sait pas quoi faire), p. ex. :
   - la division par 0 ou les instructions inconnues
   - la violation de privilège (accès interdit en mode **user**)
   - les erreurs d'accès mémoire (segmentation fault)

Les contrôleurs de périphériques peuvent interrompre les programmes pour réaliser des actions urgentes sur les périphériques

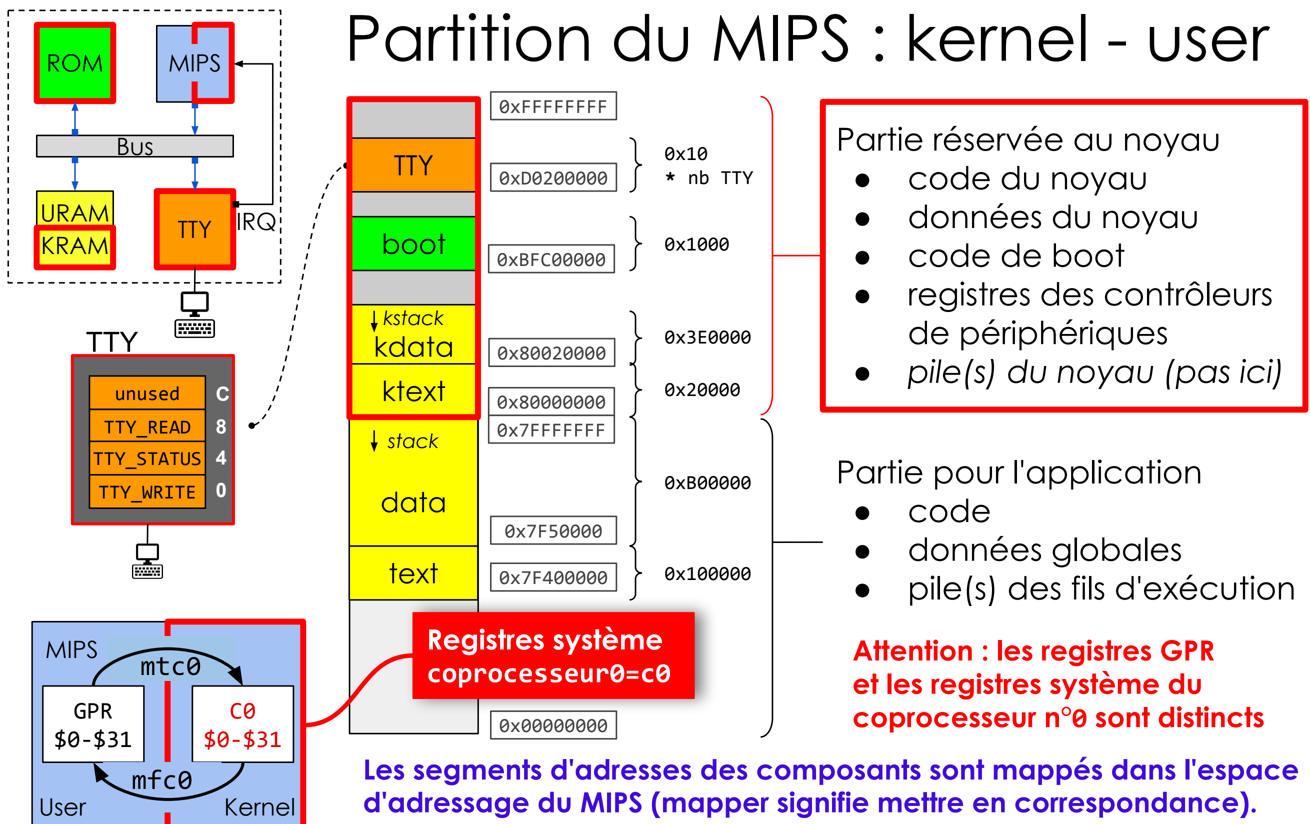
3. Le **gestionnaire d'interruption** gère les requêtes d'interruption (IRQ) (toujours attendues) venant des périphériques, p. ex. :
   - les fins de commande (lorsque le travail est fait)
   - les **ticks** d'horloge (pour que l'OS compte le temps)

Pour les 3 gestionnaires, c'est le même point d'entrée, nommé **kentry**, et placé à l'adresse `0x80000180` de l'espace d'adressage

Plan de la séance

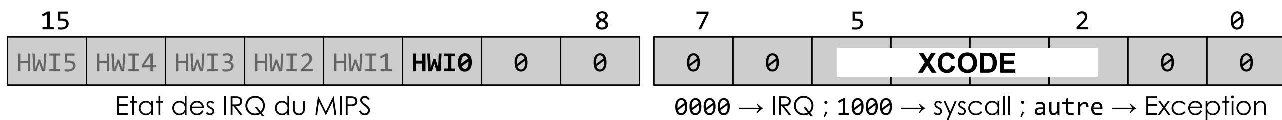
- Rappel des modes d'exécution du MIPS du point de vue matériel
- Interruptions du point de vue matériel
- Gestionnaire d'interruptions du noyau

Modes d'exécution du MIPS

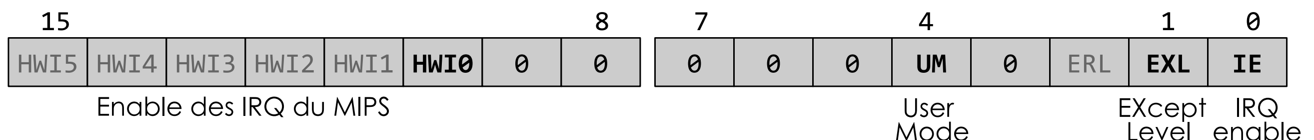


Registres système : Status, Cause, EPC

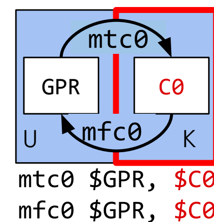
Le registre `c0_cause ($13)` contient la cause d'entrée dans le noyau (si IRQ, syscall ou except)



Le registre `c0_sr ($12)` contient le mode d'exécution du MIPS et les autorisations d'IRQ



Le registre `c0_epc ($14)` contient l'adresse de retour si c'est une IRQ
ou l'adresse de l'instruction courante pour syscall et toutes les exceptions



`mtc0 $GPR, $C0`
`mfc0 $GPR, $C0`

Entrée et sortie du noyau

`syscall` ou **exception** ou **interruption**

`c0_sr.EXL` ← 1
mise à 1 du bit EXL du registre Status Register donc passage en mode kernel interruptions masquées

`c0_cause.XCODE` ← numéro de cause
par exemple 8 si la cause est syscall

`EPC` ← `PC` ou `PC+4`
`PC` adresse de l'instruction courante pour syscall et exception
`PC+4` adresse suivante pour interruption

`PC` ← `0x80000180`
C'est là que se trouve l'entrée du noyau `kentry` toute cause confondue [syscall, exception, interruption]

`eret`

`c0_sr.EXL` ← 0
mise à 0 du bit EXL du registre Status Register donc passage en mode `c0_sr.UM` et avec interruption ou pas suivant `c0_sr.IE`
`c0_sr.UM = 1 ⇒ mode user`
`c0_sr.IE = 1 ⇒ int autorisées`

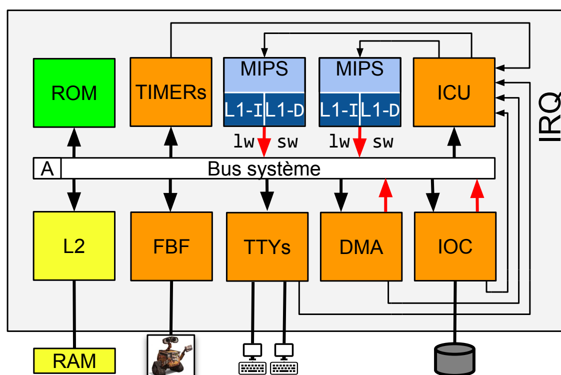
`PC` ← `EPC`
désigne l'adresse de la prochaine instruction à exécuter

Les registres du coprocesseur 0 (`c0`) (dits registres système) sont en rouge

Interruptions du point de vue matériel

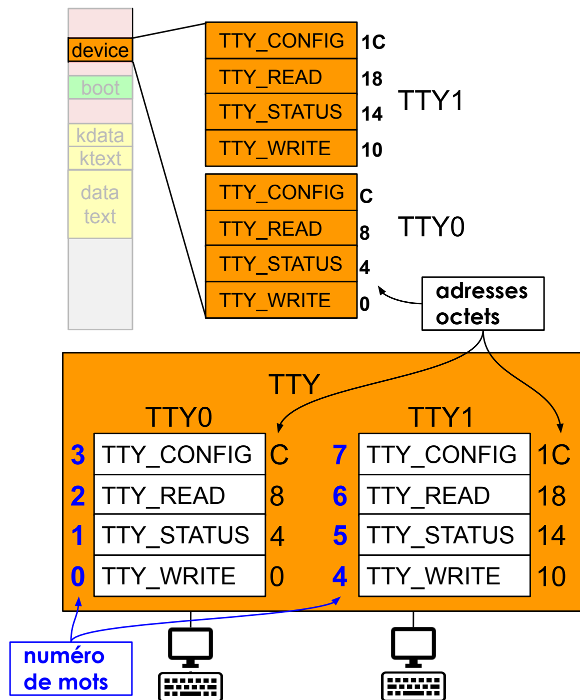
Concept d'interruption

- Une interruption, c'est la suspension de l'exécution de l'application en cours sur le processeur pour accomplir un service de plus haute priorité (dans le kernel)
- **On interrompt un programme et non un processeur, car ce dernier continue à travailler**
- Ce sont les périphériques qui font des **requêtes d'interruption** au processeur.
- Une requête d'interruption (**IRQ** pour Interrupt ReQuest) est transmise par un signal électrique à 2 états : un état inactif (baissé) et un état actif (levé).
- On dit qu'on lève une **IRQ** ou qu'on active une **IRQ**, c'est un état (non transitoire)
- Une **IRQ** doit **rester levée**/active **tant qu'elle n'est pas traitée** par le kernel.



- Le contrôleur de TTY a autant de signaux IRQ qu'il y a de TTY (`almo1` en a jusqu'à 4)
- Il faut autant de TIMER que de MIPS (`almo1` en a jusqu'à 8), chaque TIMER peut lever une IRQ périodique
- Le composant DMA (memcpy) a une IRQ
- Le contrôleur de disque IOC a une IRQ
- Les IRQ transitent par l'ICU (Interrupt Controller Unit) chargé de masquer les IRQ non désirées et de les router vers le MIPS concerné.

Contrôleur de terminaux TTY



Tous les registres sont alignés sur des mots, chaque terminal utilise un segment de 4 mots.

Pour chaque terminal

- `TTY_WRITE` 1 mot en écriture seule, le caractère ascii est mis dans l'octet de poids faible → sortie vers l'écran
- `TTY_STATUS` 1 mot en lecture seule, ≠ 0 s'il y a un caractère ascii en attente dans TTY_READ
- `TTY_READ` 1 mot en lecture seule, le caractère ascii tapé est dans l'octet de poids faible **lire TTY_READ acquitte l'IRQ du TTY concerné**
- `TTY_CONFIG` inutilisé dans cette version, mais permet la configuration p. ex. du débit d'échange avec le terminal externe.

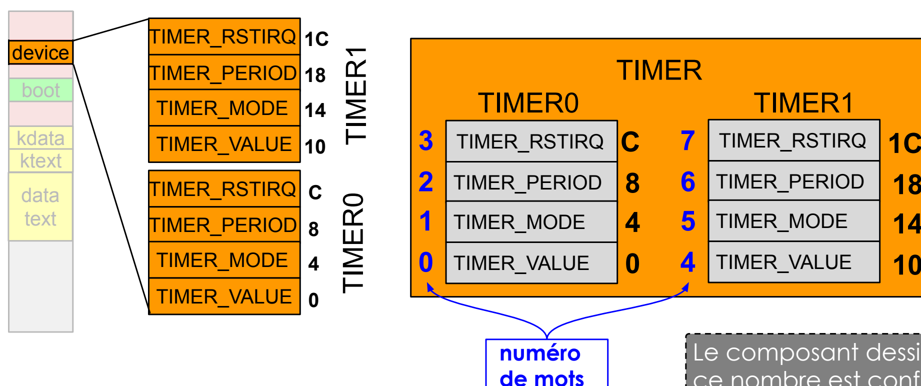
Chaque TTY lève une IRQ si un caractère est reçu par le TTY donc si son STATUS est ≠ de 0 (n TTY → n IRQ)

Le composant dessiné contient 2 TTYs, ce nombre est configurable de 1 à 4

TIMER

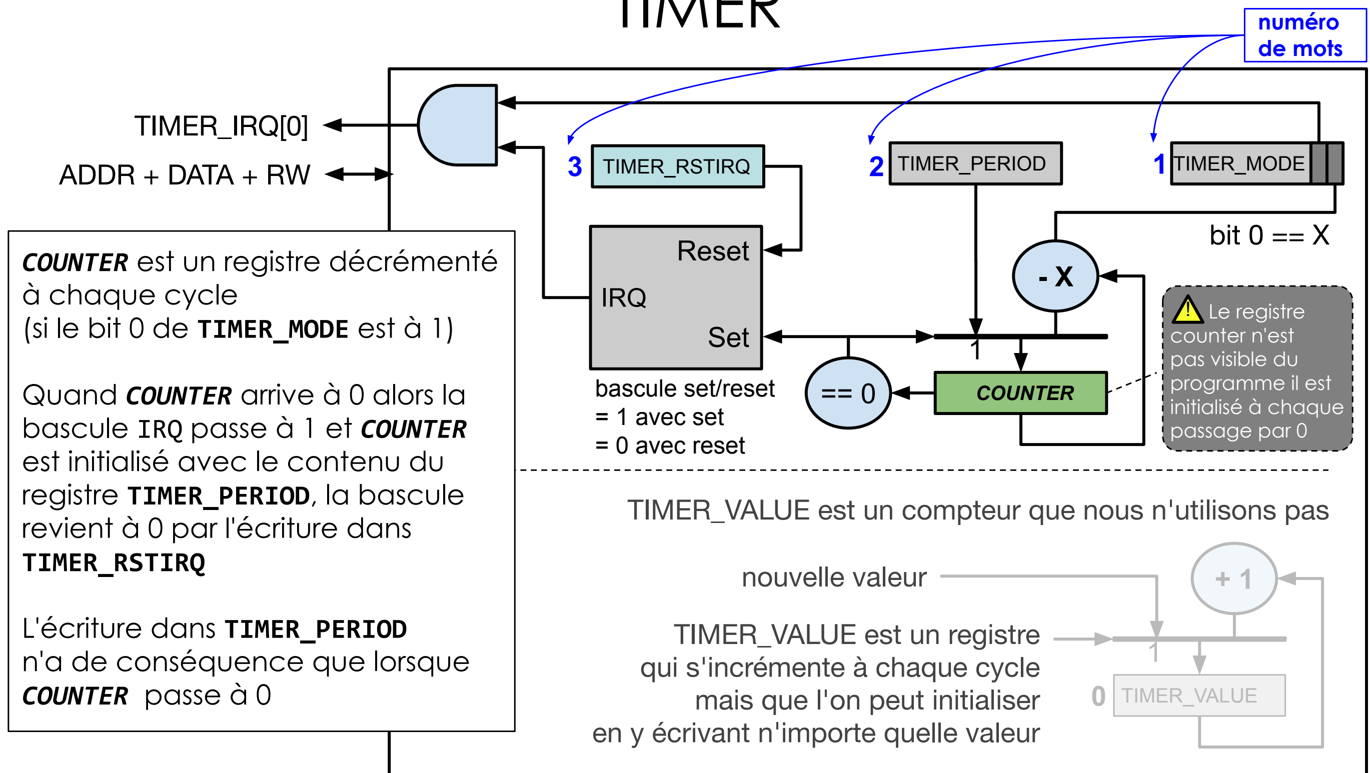
Le TIMER contient des compteurs de cycles qui peuvent lever des interruptions périodiques. C'est un périphérique cible contrôlé par des accès en lecture / écriture dans ses registres.

- 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 demandée entre 2 IRQ
- TIMER_RESETIRQ (écriture seule) **écrire n'importe quoi à cette adresse acquitte l'IRQ**



Le composant dessiné contient 2 TIMERs, ce nombre est configurable de 1 à 8

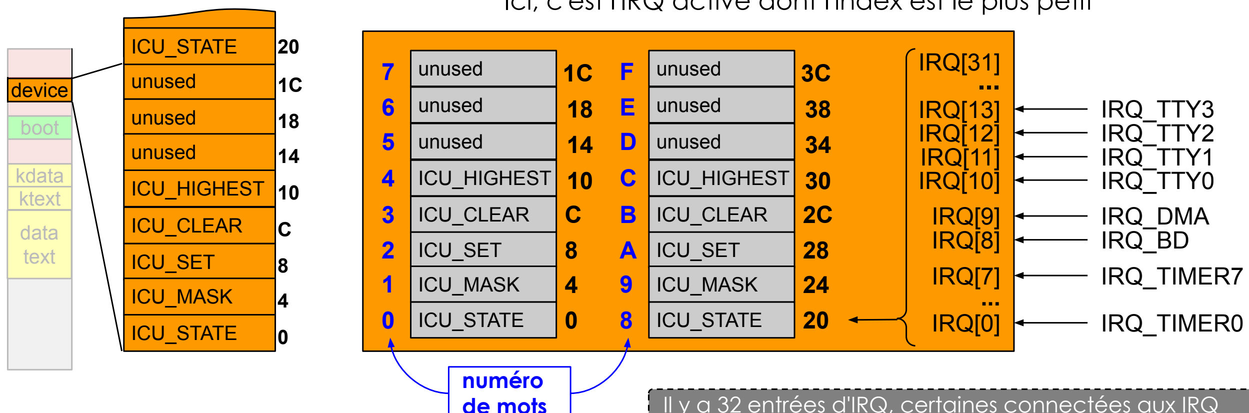
TIMER



Interrupt Controller Unit ICU

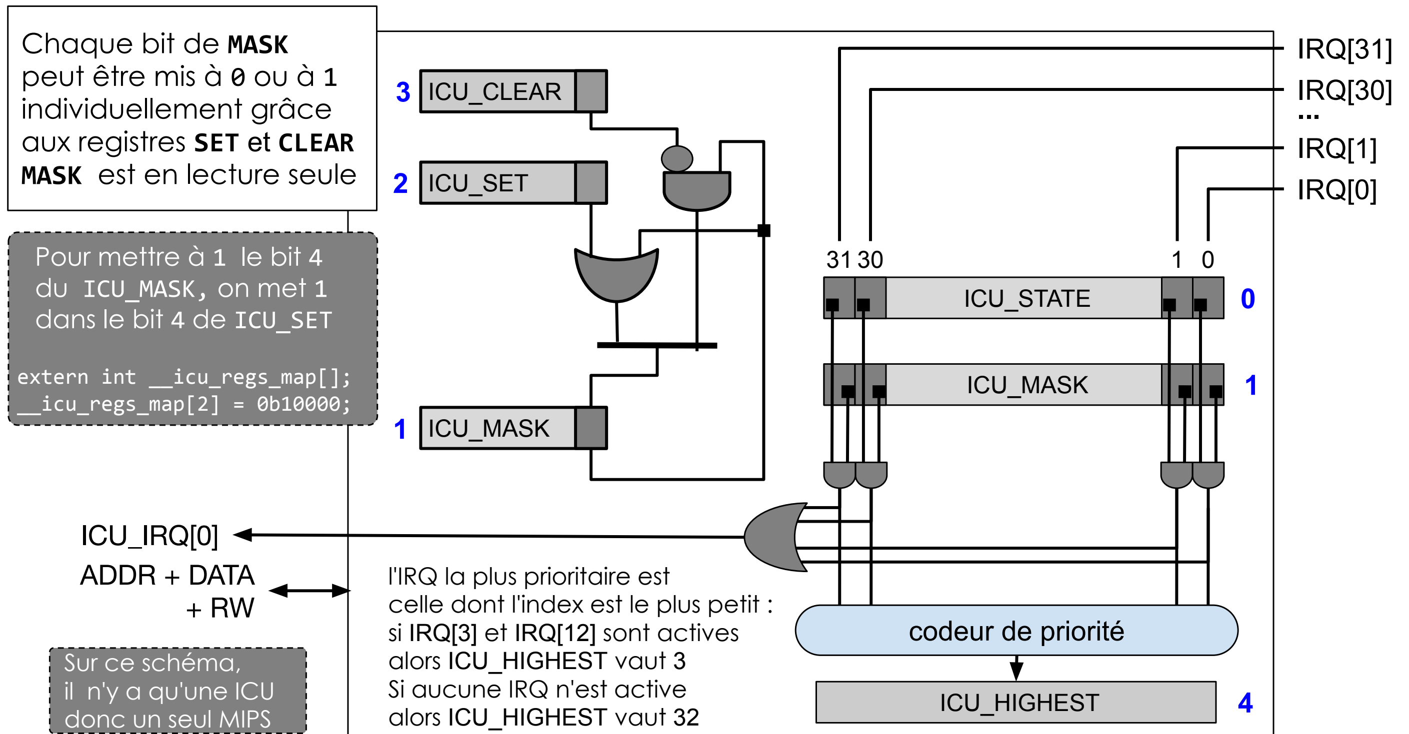
L'ICU est un concentrateur et un routeur de signaux d'IRQ. Chaque IRQ peut être masquée.
Il y a autant de jeu de registres que de MIPS (ici 2), chaque MIPS sélectionne les IRQ qu'il souhaite

- ICU_STATE (lecture seule) état des lignes IRQ (identique pour toutes les instances)
- ICU_MASK (lecture seule) masques des lignes IRQ (sélection des IRQ désirées par MIPS)
- 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_HIGHEST (lecture seule) **numéro de la ligne IRQ active et non masquée la plus prioritaire**
  Ici, c'est l'IRQ active dont l'index est le plus petit



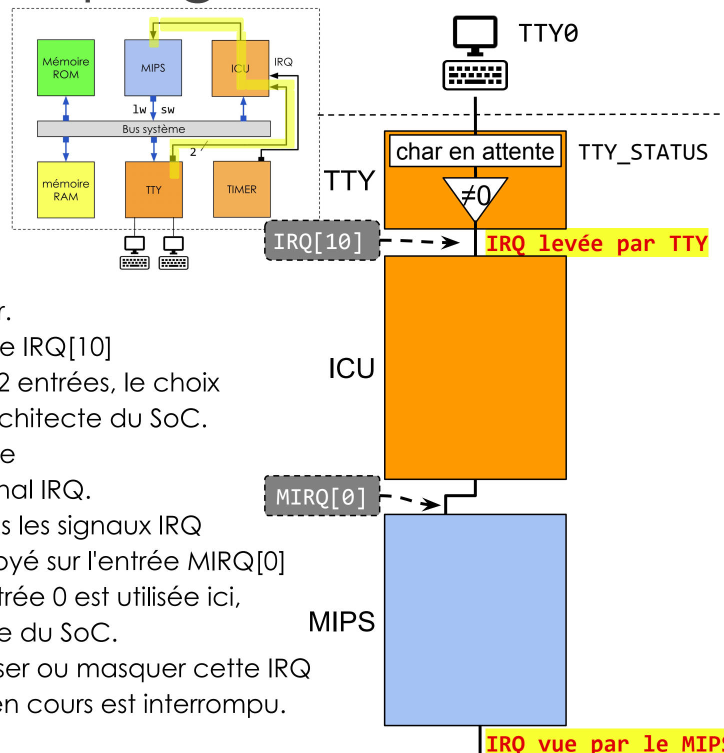
Il y a 32 entrées d'IRQ, certaines connectées aux IRQ émises par les périphériques, c'est un choix de l'architecte, les numéros sur le schéma sont ceux utilisés par almo1

Interrupt Controller Unit ICU



Niveau de masquage des IRQ

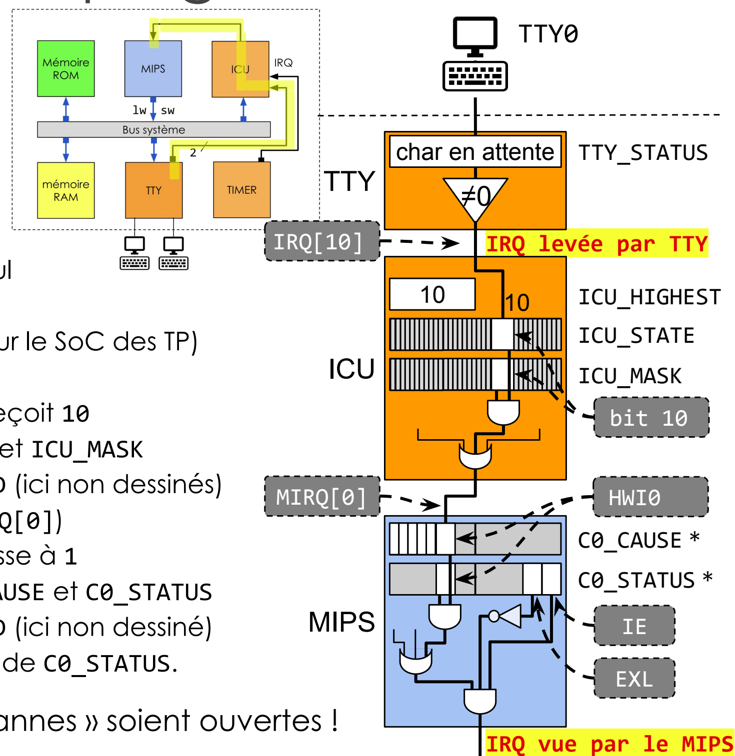
Ces schémas représentent le cheminement d'un signal IRQ depuis sa source(ici le contrôleur TTY0) jusqu'au signal vu par le MIPS.



- Une IRQ est levé par le contrôleur du TTY0 lorsqu'une touche est frappée sur le clavier.
- Le signal IRQ du TTY0 est envoyé sur l'entrée IRQ[10] du composant ICU. Le composant ICU a 32 entrées, le choix de l'entrée 10 est un choix arbitraire de l'architecte du SoC.
- L'ICU peut être configuré par le programme pour laisser passer ou pour masquer ce signal IRQ.
- l'ICU produit un signal IRQ qui combine tous les signaux IRQ qu'il reçoit et ce signal IRQ produit est envoyé sur l'entrée MIRQ[0] du MIPS (le MIPS a 6 entrées IRQ, seule l'entrée 0 est utilisée ici, c'est aussi un choix arbitraire de l'architecte du SoC.
- Le MIPS peut est configuré pour laisser passer ou masquer cette IRQ S'il n'est pas masqué, alors le programme en cours est interrompu.

Niveau de masquage des IRQ

Une IRQ est émise par un contrôleur de périphérique peut être masquée par le noyau lorsqu'il exécute du code *critique*



En TP, lors d'une frappe du clavier `TTY0` :

→ Le registre `TTY_STATUS` de `TTY0` devient non nul
→ Le contrôleur de `TTY` lève son `IRQ`
→ Le signal entre par la pin `IRQ[10]` de l'`ICU` (pour le SoC des TP)
→ Le bit `10` de `ICU_STATE` passe à `1`
→ Si c'est la seule `IRQ`, le registre `ICU_HIGHTEST` reçoit `10`
→ l'`ICU` fait un `AND` entre les bits `10` de `ICU_STATE` et `ICU_MASK`
→ puis un `OU` avec toutes les autres sorties de `AND` (ici non dessinés)
→ L'`IRQ` en sortie de l'`ICU` entre dans le MIPS (`MIRQ[0]`)
→ Le bit `HWI0` du registre de cause `C0_CAUSE` passe à `1`
→ Le MIPS fait un `AND` entre les bits `HWI0` de `C0_CAUSE` et `C0_STATUS`
→ puis un `OU` avec toutes les autres sorties de `AND` (ici non dessiné)
→ enfin on fait un `AND` avec les bits `IE` et `not EXL` de `C0_STATUS`.

Pour voir une `IRQ`, il faut que toutes les « vannes » soient ouvertes !

* `c0_cause` et `c0_status` sont détaillés plus loin

Ce qu'il faut retenir

- Les contrôleurs de périphériques permettent d'accéder aux périphériques grâce à des registres mappés dans l'espace d'adressage du MIPS
- Les IRQ (requêtes d'interruption) sont des signaux d'état émis par les contrôleurs de périphérique pour informer de la survenue d'un événement (la fin d'une commande ou l'arrivée d'une donnée)
- Une IRQ a deux états : active (ou levée), inactive (ou baissée).
- Lorsqu'une IRQ est traitée, il faut l'acquitter en accédant au contrôleur de périphérique qui la activée pour lui demander de la désactiver;
- Le composant ICU (Interrupt Controller Unit) combine les IRQ de tous les contrôleurs de périphérique pour en produire une par MIPS envoyée sur l'entrée IRQ du MIPS concerné.
- Les IRQ sont toujours attendues, mais elles peuvent être masquées pour chaque MIPS, temporairement ou définitivement, c.-à-d. ne pas être visibles (mais elles restent actives).
- Les IRQ peuvent être masquées par l'ICU (grace au registre `ICU_MASK`) ou par les MIPS eux-mêmes (grace à leur registre `c0_SR`).
- Lorsque l'ICU reçoit plusieurs IRQ actives, il détermine celle prioritaire dont il met le numéro dans le registre `ICU_HIGHEST` (n° de 0 à 31 puisque l'ICU gère 32 IRQ), Il y a autant de `ICU_HIGHEST` que de MIPS dans le SoC.

Gestionnaire d'interruption

passage application → kernel

Il y a **3 gestionnaires** d'appel du kernel : **syscall**, **exception** et **interruption**
Dans tous les cas, le MIPS saute à **kentry** en `0x80000180` avec la cause dans `c0_cause`

Le **gestionnaire des syscalls**

syscall : service demandé au kernel avec la convention d'appel ci-dessous

- `$2` contient un numéro de service (numéros communs kernel / user)
- `$4, $5, $6, $7` contiennent les arguments du service (jamais plus de 4 arguments)
- `au retour $2` contient la valeur de retour (en général `0` si tout va bien)
- seuls les registres GPR persistants (`$16` à `$23`) sont garantis inchangés
- l'instruction `syscall` se comporte presque comme un appel de fonction, sauf que la fonction appelante de `syscall` ne réserve pas de place dans la pile pour les arguments (`$4` à `$7`)

L'instruction `syscall` provoque l'appel de fonction `SYSCALL_VECTOR[$2]($4,$5,$6,$7,$2)`
`SYSCALL_VECTOR[]` est un tableau de pointeurs sur des fonctions dans le noyau

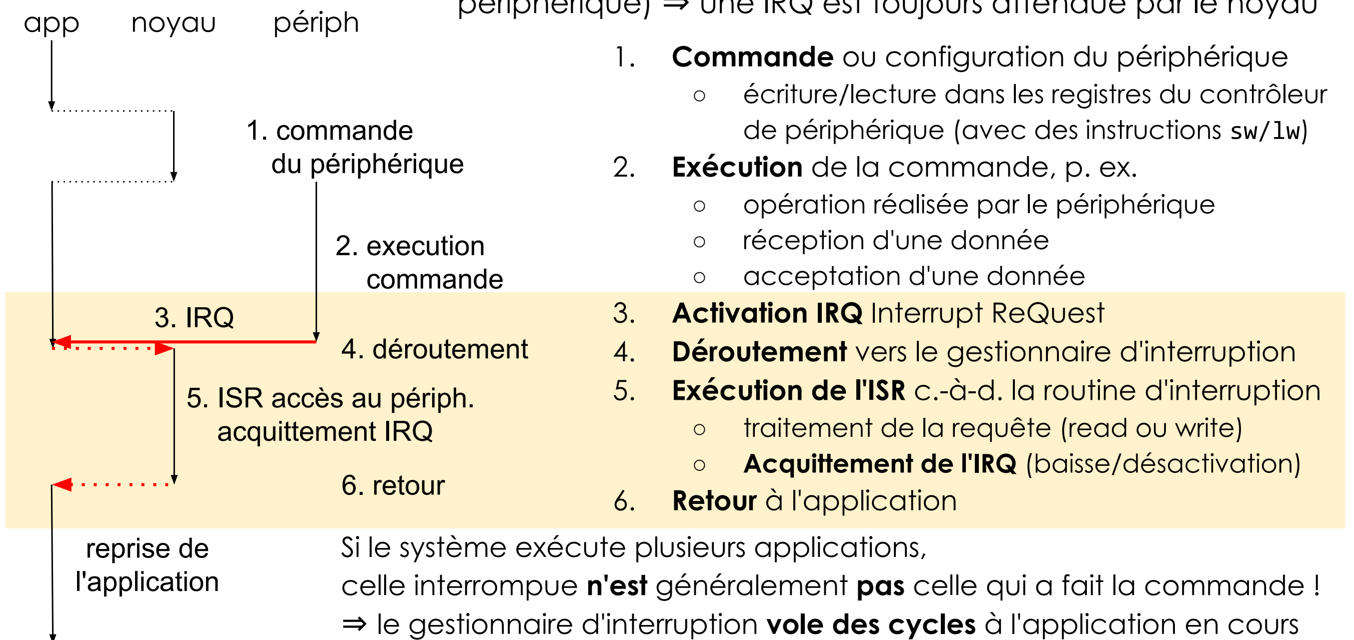
Le **gestionnaire des interruptions**

exceptions et **interruptions**

- Une exception est une faute du programme, dans notre cas, elles sont fatales, mais parfois on revient dans l'application. Ici, on affiche les registres et on se bloque.
- Les interruptions sont demandées par les périphériques, elles s'insèrent entre 2 instructions. Dans les deux cas, tous les registres sont conservés intacts, l'interruption a juste « volé » du temps à l'application courante.

Déroulement d'une interruption

Une interruption est la conséquence d'une cause (la commande ou la configuration d'un périphérique) ⇒ une IRQ est toujours attendue par le noyau



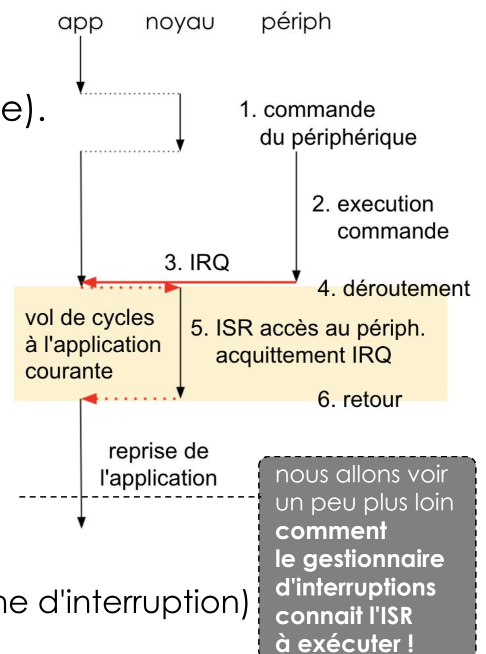
1. **Commande** ou configuration du périphérique
   - écriture/lecture dans les registres du contrôleur de périphérique (avec des instructions `sw/lw`)
2. **Exécution** de la commande, p. ex.
   - opération réalisée par le périphérique
   - réception d'une donnée
   - acceptation d'une donnée
3. **Activation IRQ** Interrupt ReQuest
4. **Déroutement** vers le gestionnaire d'interruption
5. **Exécution de l'ISR** c.-à-d. la routine d'interruption
   - traitement de la requête (read ou write)
   - **Acquittement de l'IRQ** (baisse/désactivation)
6. **Retour** à l'application

Si le système exécute plusieurs applications,
celle interrompue **n'est** généralement **pas** celle qui a fait la commande !
⇒ le gestionnaire d'interruption **vole des cycles** à l'application en cours

Gestionnaire d'interruptions

Le gestionnaire d'interruption est invoqué (3.)
lorsqu'**une IRQ s'active** (et qu'elle n'est pas masquée).
Étapes de traitement :

- **déroutement** (4.) vers **kentry**
  `c0_EPC ← PC+4 ; c0_sr.EXL ← 1 ; c0_cause.XCODE ← 0`
  `PC ← 0x80000180`
- analyse du champ `XCODE` du registre `c0_cause`
- appel du gestionnaire d'interruption
- sauvegarde des registres temporaires
- lecture du numéro de l'IRQ dans ICU_HIGHEST
- **Exécution de l'ISR associée à ce numéro d'IRQ** (5.)
  - accès aux registres du périphérique
  - acquittement de l'IRQ (c.-à-d. baisser la ligne d'interruption)
- restauration des registres temporaires
- **retour** (6.) au programme interrompu



ISR : Interrupt Service Routine

Un Pilote de périphérique (Device Driver) regroupe l'ensemble des fonctions d'accès à un contrôleur de périphérique. On y trouve :

- les fonctions de commandes (p. ex. pour le moment : `tty_puts()`, `tty_gets()`)
- et **une ISR pour gérer la terminaison des commandes** (p. ex: `tty_isr()`)

Nous verrons après que le noyau définit une API dans la HAL pour uniformiser les pilotes

Les ISR (ou routines d'interruption) sont les fonctions qui traitent les IRQ

- Elles accèdent aux registres du contrôleur de périphérique ayant levé l'IRQ
  Cette étape est spécifique à chaque périphérique
- *Elles peuvent aussi programmer une nouvelle commande dans le cas où il y a une file d'attente de commandes envoyées par les applications et qui n'ont pu être démarrées parce que le périphérique était occupé.*
- *Elles peuvent demander au noyau de changer l'état de l'application qui était en attente de la terminaison de la commande*
- Elles acquittent l'IRQ en accédant aux registres du contrôleur de périphériques. Cette étape est spécifique à chaque périphérique
- Pour l'OS des TP, les ISR ne sont pas interruptibles → c'est un choix simplificateur

pas dans la première version de l'OS

Sélection et appel de la bonne ISR

Ce schéma représente :

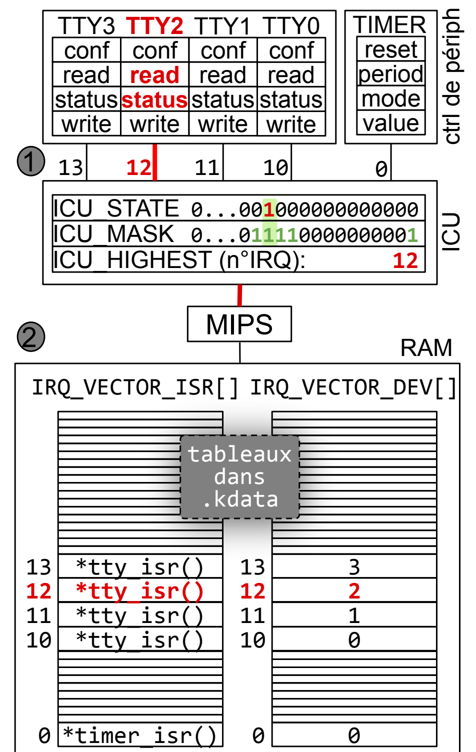
1. des registres de contrôleurs de périphériques du SoC `almo1` impliqués dans le traitement des IRQ et
2. le vecteur d'interruption qui permet au noyau de savoir quelle ISR exécuter en fonction du numéro de l'IRQ active

Le registre système `c0_SR` du MIPS n'est pas représenté ici parce qu'on suppose que l'IRQ n'est pas masquée, et donc les bits `c0_SR.HWI0` et `c0_SR.IE` contiennent nécessairement `'1'`

On suppose que l'utilisateur frappe sur une touche de `TTY2` :

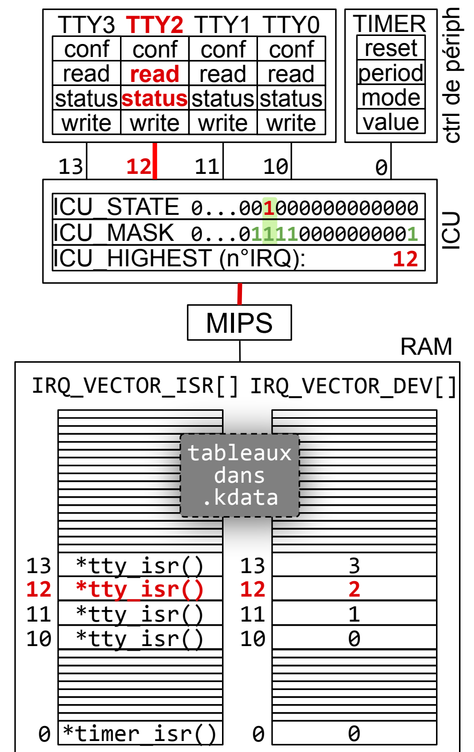
- `TTY_READ` de `TTY2` contient le code ascii de la touche
- `TTY_STATUS` de `TTY2` prend une valeur non nulle
- l'IRQ de `TTY2` s'active, elle est branchée sur l'entrée 12 de l'ICU (c'est le choix de l'architecte du SoC)
- le bit 12 de `ICU_STATE` passe à `1`
- le registre `ICU_HIGHEST` prend la valeur `12`

Nous allons voir maintenant l'usage du vecteur d'interruption.



Sélection et appel de la bonne ISR

- Lorsqu'une IRQ non masquée se lève, le MIPS est dérouté vers **kentry**
  `PC←0x80000180` et `c0_epc← PC+4`, `c0_sr.EXL←1` et `c0_cause.XCODE←0`,
- Ici, `TTY2` active son IRQ connectée sur l'entrée `12` de l'`ICU` et comme le bit `12` de `ICU_MASK` est à `1` alors `ICU_HIGHEST` prend la valeur `12`
- **kentry** appelle le **gestionnaire d'interruption** qui sauve les registres les registres temporaires avant d'**appeler la bonne ISR**
- Le gestionnaire utilise un tableau `IRQ_VECTOR_ISR[]` indexé par le numéro d'IRQ, dont les cases contiennent les pointeurs vers les `ISR`
- Il n'y a qu'une fonction `ISR()` par type de périphérique, donc il n'y a qu'une fonction `tty_isr()` utilisée quel que soit le numéro de TTY.
- Les fonctions `ISR()` doivent savoir quelle instance a levé son IRQ, le gestionnaire utilise un autre tableau indexé par le numéro d'IRQ, `IRQ_VECTOR_DEV[]`, dont les cases contiennent le numéro d'instance, lequel est passé en argument aux fonctions `ISR()`
- Le gestionnaire d'interruption appelle donc la fonction :
  `IRQ_VECTOR_ISR[ICU_HIGHEST](IRQ_VECTOR_DEV[ICU_HIGHEST])`
- Dans l'exemple à droite, le gestionnaire appelle : `tty_isr(2)`



Configuration des IRQ

La configuration des IRQ est faite par **`arch_init()`** (qui est dans la HAL) appelée par **`kinit()`**

1. Configuration du matériel
   - Configuration de chaque composant pouvant lever des IRQ (ici: TTY et TIMER)
   - Configuration du registre MASK de l'ICU pour choisir les IRQ que l'OS veut « voir »
   - Configuration du registre **`c0_sr`** du MIPS pour autoriser les interruptions
2. Configuration du noyau
   - Liaison (appelé **binding**) des couples (n° IRQ → ISR) et (n° IRQ → n°instance) en écrivant dans les tableaux `IRQ_VECTOR_ISR[]` et `IRQ_VECTOR_DEV[]`
     Notez que dans un OS plus avancé `IRQ_VECTOR_DEV[]` contient un pointeur sur une structure de donnée propre au périphérique (structure « `device` »)

harch.c

```
void arch_init (int tick) {
    timer_init (0, tick);
    icu_set_mask (0, 0);
    irq_vector_isr [0] = timer_isr;
    irq_vector_dev [0] = 0;

    for (int tty = 1; tty < NTTYS; tty++) {
        icu_set_mask (0, 10+tty);
        irq_vector_isr [10+tty] = tty_isr;
        irq_vector_dev [10+tty] = tty;
    }
}

static void timer_init (int timer, int tick) {
    timer = timer % NCPUS;
    __timer_regs_map[timer].resetirq = 1;
    __timer_regs_map[timer].period =  tick;
    __timer_regs_map[timer].mode = (tick)?3:0;
}

static void icu_set_mask (int icu, int irq){
    icu = icu % NCPUS;
    __icu_regs_map[icu].set = 1 << irq;
}
```

* kentry → irq_handler → isrcall → tty_isr

U/K | Kernel

kernel/hcpua.S
```
// c0_cause.XCODE contient 0 (car IRQ)
// c0_EPC contient l'adresse de la prochaine instruction
// c0_SR.EXL est à 1 → mode kernel avec IRQ masquées
kentry:
    mfc0    $26,    $13             // $26 ← c0_CAUSE
    andi    $26,    $26,    0x3C    // $26 ← XCODE * 4
    li      $27,    0x20            // $27 ← 8 * 4 (syscall)
    beq     $26,    $27,    syscall_handler // si syscall
    beq     $26,    $0,     irq_handler     // si IRQ
    j       kpanic                  // exception
syscall_handler:
    // code du gestionnaire de syscall
irq_handler:
    // 20 regs to save (17 tmp regs+HI+LO+$31)
    addiu   $29,    $29,    -20*4
    sw      $31,    19*4($29)       // $31 lost by jal
    // save 17 tmp reg: $1 à $15, $24, $25
    sw      $1,     1*4($29)
    [...]
    sw      $25,    17*4($29)
    mflo    $2                      // get LOW
    mfhi    $3                      // get HIGH
    sw      $2,     19*4($29)       // save LOW
    sw      $3,     0*4($29)        // save HIGH
    jal     isrcall                 // call the right ISR
```
suite

Vecteur d'interruption utilisé par `isrcall()` pour appeler la bonne ISR sur le bon périph.

	IRQ_VECTOR_ISR[]		IRQ_VECTOR_DEV[]
31		31	
13	*tty_isr()	13	3
12	*tty_isr()	12	2
11	*tty_isr()	11	1
10	*tty_isr()	10	0
0	*timer_isr()	0	0

kernel/harch.c
```
void isrcall (void) {
    int irq = icu_get_highest (cpuid());
    irq_vector_isr[irq] (irq_vector_dev[irq]);
}
```

kernel/harch.c
```
void tty_isr (int tty) {
    int c = __tty_regs_map[tty].read;   // get char from tty
    __tty_regs_map[tty].write = c;      // loopback to tty
    [...]
}
```

kernel/hcpua.S
```
    lw      $3,     0*4($29)            // get HIGH
    lw      $2,     18*4($29)           // get LOW
    mthi    $3                          // restore HIGH
    mtlo    $2                          // restore LOW
    // restore 17 tmp reg: $1 à $15, $24, $25, HI, LO
    lw      $25,    17*4($29)
    [...]
    lw      $1,     1*4($29)
    lw      $31,    19*4($29)           // restore $31
    addiu   $29,    $29, 20*4           // restore the stack ptr
    eret                                // j EPC & c0_SR.EXL <- 0
```


En résumé

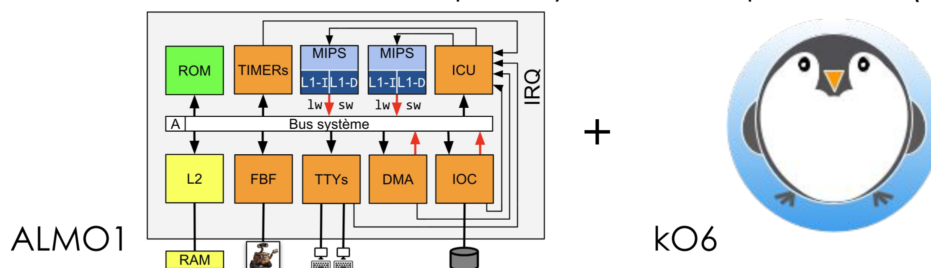
- L'entrée dans le noyau est `kentry` à l'adresse `0x80000180` quelque soit la cause d'appel (`syscall`, `exception`, `interruption`), `kentry` analyse `c0_cause.XCODE` puis appelle vers le bon gestionnaire
- Quand une IRQ non masquée est levée, elle provoque l'interruption du programme en cours et l'entrée dans le noyau qui sait grâce à `c0_cause.XCODE` que c'est une IRQ
- Pour qu'une IRQ interrompt le programme en cours, il faut :
    - qu'elle soit levée (activée) par le contrôleur de périphérique
    - qu'elle ne soit pas masquée par l'ICU (grâce à son registre `ICU_MASK`)
    - qu'elle ne soit pas masquée par le processeur (grâce à son registre `c0_status`)
- Le gestionnaire d'interruption lit l'ICU pour connaître le numéro de l'IRQ (`ICU_HIGHEST`) qu'il utilise comme index dans le vecteur d'interruption pour récupérer l'adresse des l'ISR et le numéro d'instance du device et appeler la fonction **`IRQ_VECTOR_ISR[ICU_HIGHEST](IRQ_VECTOR_DEV[ICU_HIGHEST])`**
- l'ISR accède au périphérique concerné et acquitte l'IRQ pour baisser le signal

Conclusion

- Nous avons abordé
  - les modes d'exécution du MIPS et les raisons du passage de mode
  - les requêtes d'interruptions et leur cheminement dans le SoC
  - le gestionnaire des interruptions
- Dans le prochain cours, nous verrons une version simplifiée des threads

Travaux Pratiques

Le but des TME, c'est que vous compreniez bien ce qu'il y a dans un SoC (almo1) et comment fonctionne un petit système d'exploitation (kO6).



- Pour le 2e TME, nous allons entrer dans le code de kO6 pour répondre à un grand nombre de petites questions dans le but de vous l'approprier.
- Vous allez ajouter une ISR pour le DMA