Interruptions

LU3INx29 Architecture des ordinateurs 1

franck.wajsburt@lip6.fr

4 déc 2023

Ce que nous avons vu

- Le MIPS propose 2 modes d'exécution : `kernel` et user
  - Le mode `kernel` est utilisé par le kernel pour gérer les ressources de la machine (le/s processeur/s, la mémoire et les périphériques)
  - Le mode `user` est utilisé par les applications, ce mode interdit l'accès à une partie de l'espace d'adressage et à certaines instructions
- Le kernel démarre une application en sautant (avec `eret`) à la fonction `_start()` placée par convention au début de la section `.text`
- L'application revient dans le kernel pour 3 raisons :
  - Les **appels système** avec l'instruction `syscall` pour demander un **service**
  - Les **exceptions** quand l'application exécute une **instruction incorrecte**
  - Les **interruptions** quand un **périphérique demande le processeur** pour exécuter un traitement urgent → nous allons voir comment !

Questions pour cette séance

Le programme en cours est interrompu pour exécuter
une opération urgente demandée par un contrôleur de périphérique.

- Comment les interruptions sont-elles demandées ?
  → Un contrôleur de périphérique fait une requête d'interruption : IRQ
- Comment le kernel fait-il pour les traiter ?
  → Le programme est interrompu pour exécuter le gestionnaire d'interruptions
- Que signifie : « *une IRQ provoque l'exécution d'une ISR* » ?
  → Le gestionnaire d'interruptions exécute des routines (ISR) propres à chaque IRQ

Plan de la séance

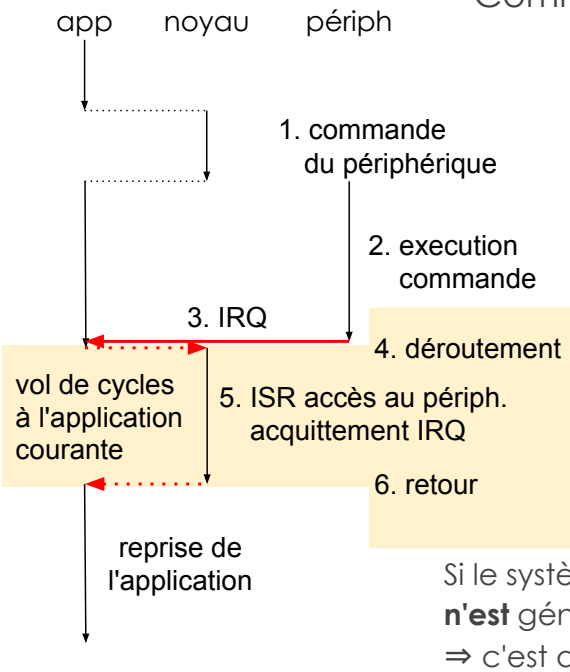
- Qu'est-ce qu'une interruption ?
- Interruptions vues du matériel
- Interruptions vues du logiciel

Qu'est-ce qu'une interruption ?

- Une interruption, c'est la suspension de l'exécution de l'application en cours sur le processeur pour accomplir une opération de plus haute priorité (un service) ou pour changer de ***thread*** * ou même pour changer d'application.
- Les contrôleurs de périphériques font des **requêtes d'interruption** au processeur.
- Le terme requête signifie demande, ordre, appel, etc. mais on utilise requête.
- Une requête d'interruption (**IRQ** pour Interrupt ReQuest) est transmise par un simple signal électrique à 2 états : un état inactif (baissé) et un état actif (levé).
- On dit qu'on lève une **IRQ** ou alors qu'on active une **IRQ**, c'est un état (non transitoire)
- Une **IRQ** doit **rester levée**/activée **tant qu'elle n'a pas été traitée** par le kernel.
- La suspension du programme en cours permet d'exécuter une **ISR** (**I**nterrupt **S**ervice **R**outine) dans le noyau pour traiter l'**IRQ**
- L'**ISR** communique avec le périphérique qui a levé/activé l'**IRQ** au moins pour lui demander de la baisser/désactiver → On dit que l'**ISR acquitte l'IRQ**
- Un composant reçoit d'abord une commande du noyau pour faire quelque-chose et le composant signale par une IRQ que cette commande est traitée.
- Une IRQ est donc toujours attendue par le noyau puisque c'est la conséquence d'une commande → une IRQ non attendue est toujours une erreur.

** Les threads sont des fils d'exécution dans le programme*

Déroulement d'une interruption ?



Commande (cause) → interruption (conséquence)

1. **Commande** du périphérique par le noyau
   - écriture/lecture dans les registres du contrôleur de périphérique (avec des instructions `sw/lw`)
2. **Exécution** de la commande par le périphérique
   - une opération dans le périphérique
   - l'attente de réception d'une donnée
   - l'attente de l'envoi d'une donnée
3. **Activation de l'IRQ** (**I**nterrupt **R**e**Q**uest)
4. **Interruption** et **déroutement** vers le noyau
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
⇒ c'est donc un **vol de cycle** pour l'application en cours

Interruptions vues du matériel

Comment les IRQs sont routées vers le ou les MIPS ?

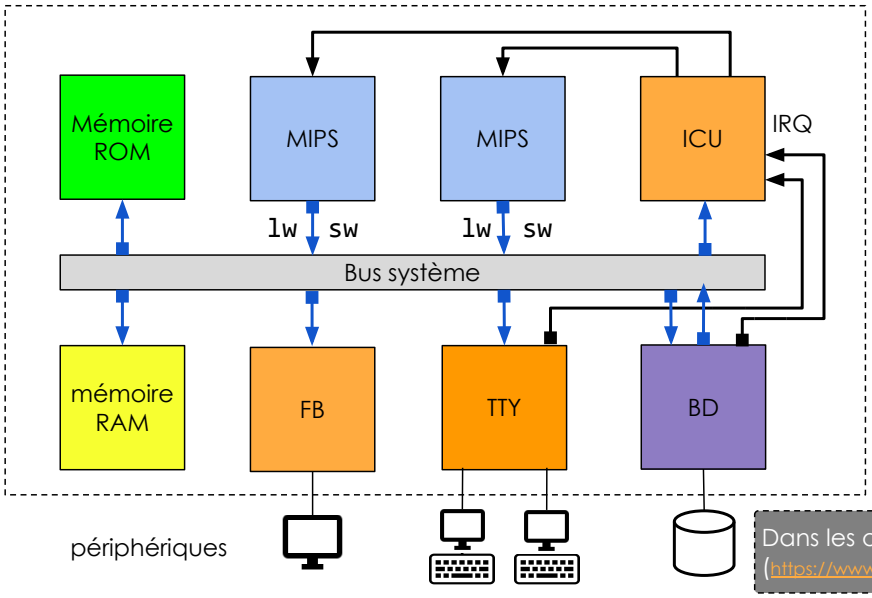
→ Il y a beaucoup d'IRQ, elles doivent être « routées » vers un processeur via un « concentrateur d'IRQ » nommée ICU…

Comment le noyau connaît le périphérique ayant levé une IRQ ?

→ C'est le « concentrateur d'IRQ » qui contient l'information sur la source…

Routage matériel des IRQ en général

Les IRQ transitent par un un composant spécial nommé **ICU**
ICU = **I**nterrupt **C**ontroler **U**nit



Dans le cas général, il y a plusieurs MIPS, en TP, il n'y en a qu'1 !

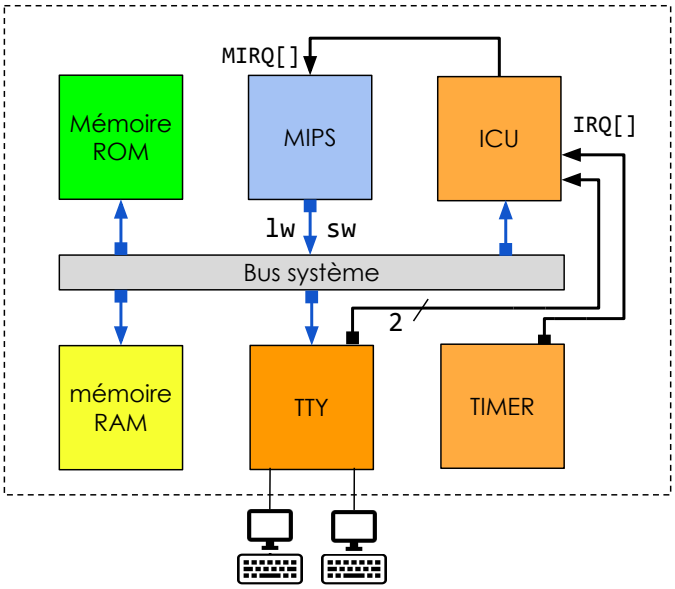
S'il y a plusieurs MIPS, chaque MIPS gère un sous-ensemble d'IRQ

Le rôle de l'ICU est de router les IRQ vers les MIPS concernés.

Dans les ordinateurs PC, l'ICU est nommé PIC (https://www.wikiwand.com/en/Programmable_interrupt_controller)

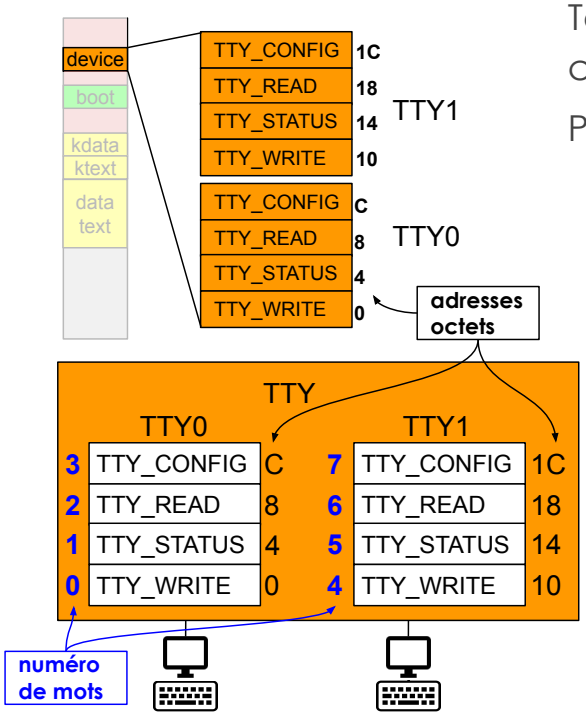
Routage matériel des IRQ pour les TPs

il y a 1 seul MIPS et seuls 2 composants peuvent lever des IRQ : TTY et Timer



- Le contrôleur de terminaux TTY a autant de signaux d'IRQ qu'il y a de TTY.
  En TP, jusqu'à 4 TTY et 2 sur le schéma
- Le composant TIMER peut lever plusieurs signaux d'IRQ périodiquement.
  En TP, il n'y a qu'1 seul TIMER ⇒ 1 seule IRQ
- L'ICU peut router **jusqu'à 32** signaux d'IRQ (pins `IRQ[]`) venant des contrôleurs de périphériques (TTY, TIMER, etc.) vers le MIPS
  l'ICU peut *masquer* chacun des signaux d'IRQ et indiquer le numéro de l'IRQ active la plus prioritaire.
  En TP, il y a 5 IRQ max (4 TTY et 1 TIMER)
- L'IRQ en sortie de l'ICU entre sur l'une des 6 entrées d'IRQ du MIPS (pins `MIRQ[]`).
  En TP, on n'utilise que l'entrée (`MIRQ[0]`)

Périphérique TTY : Contrôleur de terminaux



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

Pour chaque terminal, il y a 4 registres :

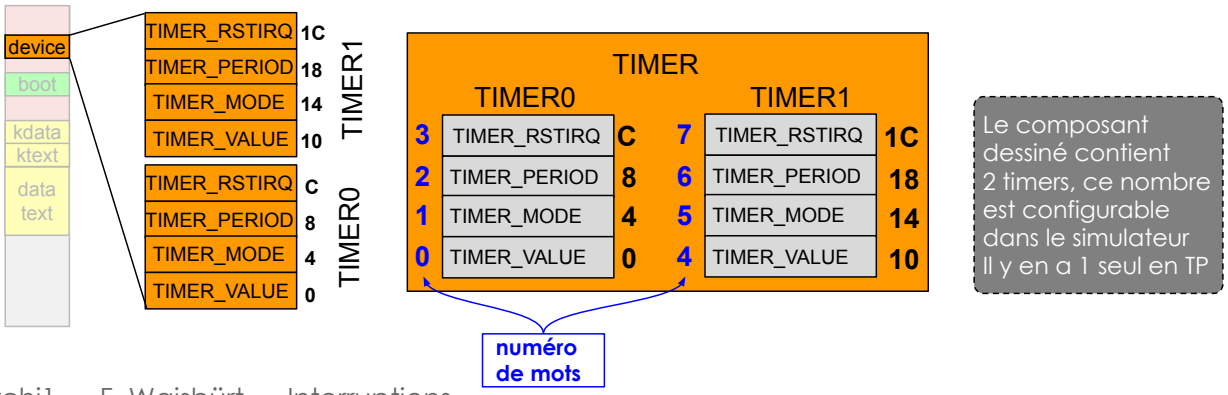
- `TTY_WRITE` 1 mot en écriture seule, le caractère ascii est mis dans l'octet de poids faible → sortie vers l'écran
- `TTY_STATUS` 1 mot en lecture seule, ≠ 0 s'il y a un caractère en attente dans TTY_READ
- `TTY_READ` 1 mot en lecture seule, le caractère tapé est dans l'octet de poids faible
- `TTY_CONFIG`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

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

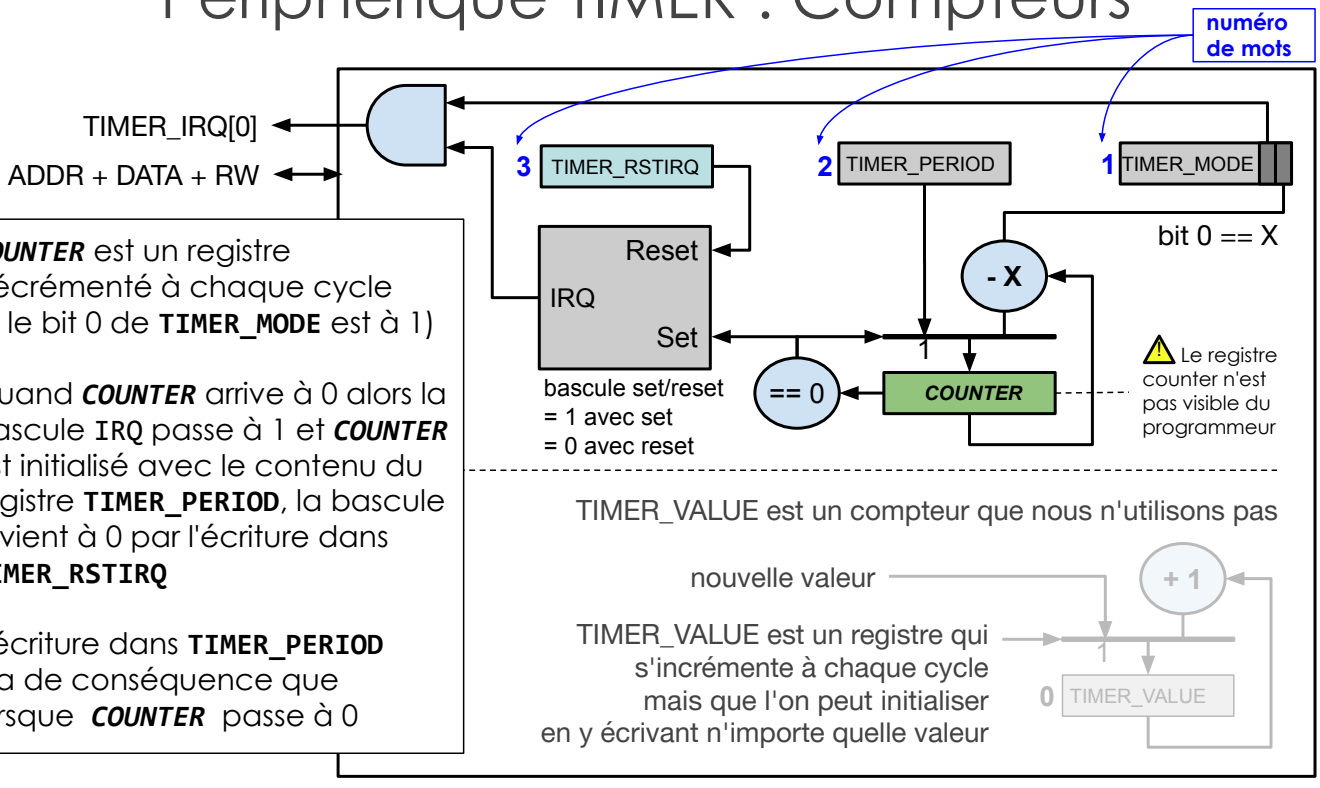
Périphérique TIMER : Compteurs

Le TIMER contient des compteurs de temps 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 entre 2 IRQ
- `TIMER_RESETIRQ` (écriture seule) **écrire à cette adresse acquitte l'IRQ**



Périphérique TIMER : Compteurs



COUNTER est un registre décrémenté à chaque cycle (si le bit 0 de **`TIMER_MODE`** est à 1)

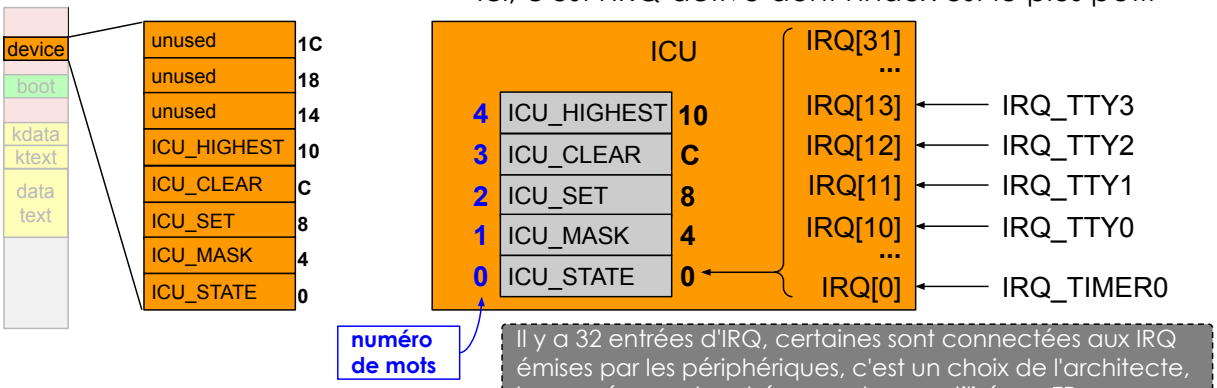
Quand ***COUNTER*** arrive à 0 alors la bascule `IRQ` passe à 1 et ***COUNTER*** est initialisé avec le contenu du registre **`TIMER_PERIOD`**, la bascule revient à 0 par l'écriture dans **`TIMER_RSTIRQ`**

L'écriture dans **`TIMER_PERIOD`** n'a de conséquence que lorsque ***COUNTER*** passe à 0

Périphérique ICU (Interrupt Controller Unit)

L'ICU est un concentrateur de signaux d'IRQ. Chaque IRQ peut être masquée.
C'est un périphérique cible contrôlé par des lectures / écritures dans ses registres.

- `ICU_STATE` (lecture seule) état des lignes IRQ
- ICU_MASK (lecture seule) masques des lignes IRQ (sélection des IRQ désirées)
- `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 la plus prioritaire**
  Ici, c'est l'IRQ active dont l'index est le plus petit



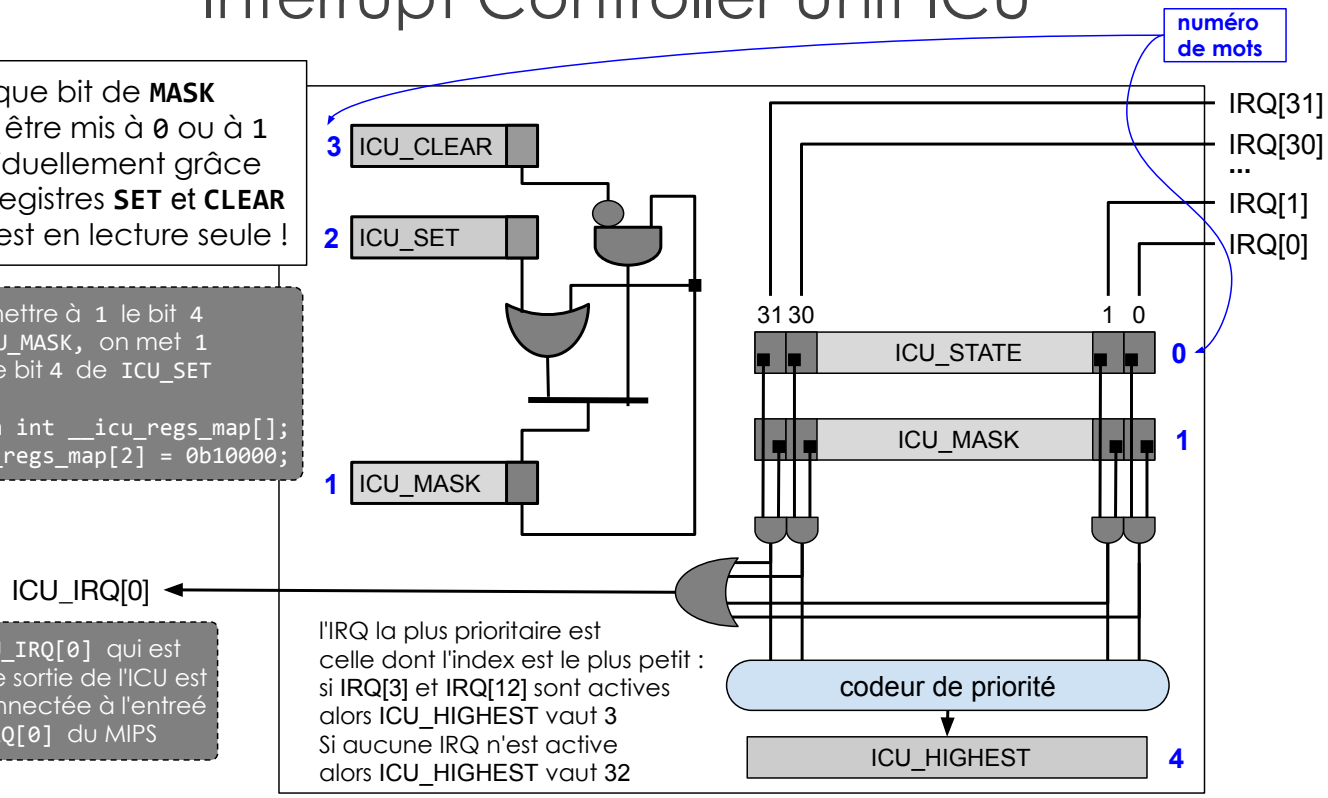
Interrupt Controller Unit ICU

Chaque bit de **MASK** peut être mis à `0` ou à `1` individuellement grâce aux registres **SET** et **CLEAR**
MASKest en lecture seule !

Pour mettre à `1` le bit `4` du `ICU_MASK`, on met `1` dans le bit `4` de `ICU_SET`

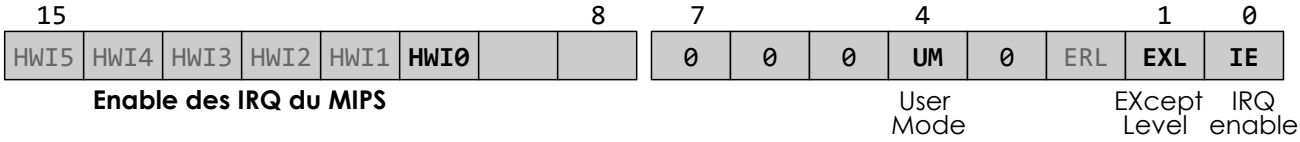
```
extern int __icu_regs_map[];
__icu_regs_map[2] = 0b10000;
```

`ICU_IRQ[0]` qui est une sortie de l'ICU est connectée à l'entreé `MIRQ[0]` du MIPS

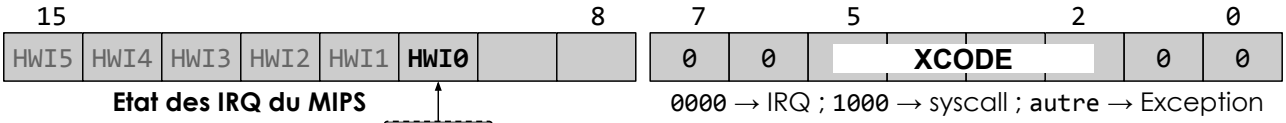


Rappel registres système : Status, Cause, EPC

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

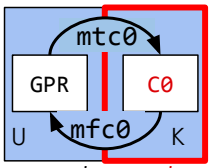


Le registre `c0_cause ($13)` contient la cause d'entrée dans le noyau (si IRQ, syscall ou except)



Le registre `c0_epc ($14)`

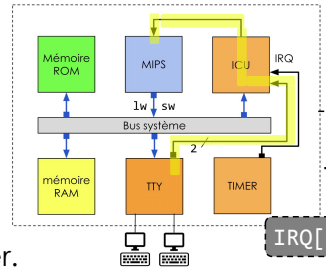
l'adresse de retour si c'est une IRQ ou sinon pour syscall et toutes les exceptions c'est l'adresse de l'instruction courante

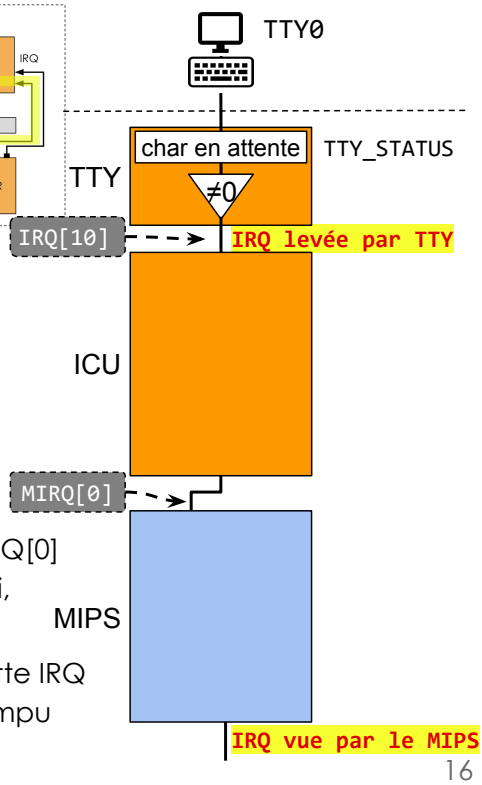



```
mtc0 $GPR, $C0
mfc0 $GPR, $C0
```


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.

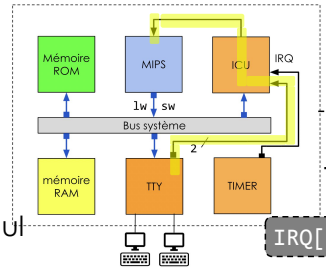


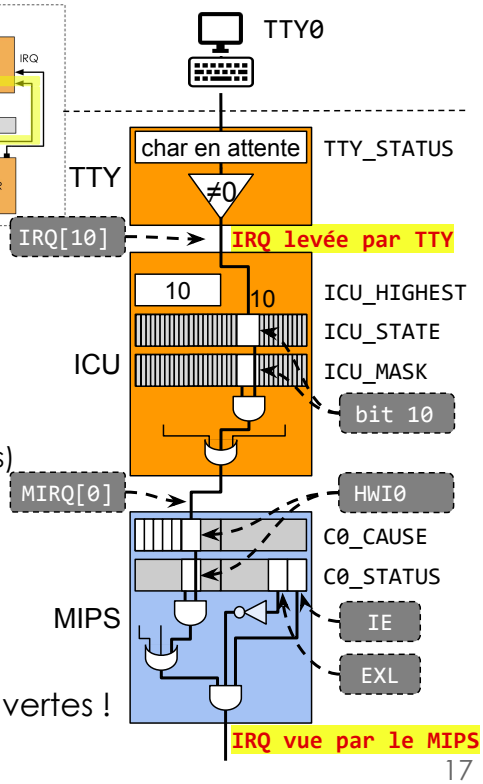


- Une IRQ est levé par le contrôleur du TTY0 lorsqu'une touche est frappée sur le clavier.
- Le signal IRQ du `TTY0` est envoyé sur l'entrée IRQ[10] du composant ICU. Le composant ICU a 32 entrées, le choix de l'entrée 10 est un choix arbitraire de l'architecte du SoC.
- L'ICU peut être configuré par 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 dessinées)
→ L'`IRQ` en sortie de l'`ICU` entre dans le MIPS (`MIRQ[0]`)
→ Le bit `HWI0` du registre de 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 !

Ce qu'il faut retenir

- Les IRQ (i.e. les requêtes d'interruption) sont des signaux d'état émis par les contrôleurs de périphérique pour informer le noyau de la survenue d'un événement tel que la fin d'une commande ou l'arrivée d'une donnée.
- Une IRQ a 2 états : actif (c.-à-d. levé) et inactif (c.-à-d. baissé).
- Lorsqu'une IRQ est traitée, il faut l'acquitter en accédant au contrôleur de périphérique qui l'a 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 seule, envoyée vers le MIPS.
- Les IRQ sont toujours attendues, mais elles peuvent être masquées, temporairement ou définitivement, c.-à-d. ne pas être visibles du MIPS.
- Les IRQ peuvent être masquées par l'ICU (grâce au registre `ICU_MASK`) ou par le MIPS lui-même (grâce au 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)

Interruptions vues du logiciel

Que doit faire le noyau lorsqu'une IRQ survient ?

→ Il sauve la valeur de tous les registres et appelle le gestionnaire d'interruption…

Comment le noyau sait quelle routine d'interruption appeler ?

→ Il dispose d'une table indexée par le numéro d'IRQ contenant les ISR…

Que doit faire une ISR ?

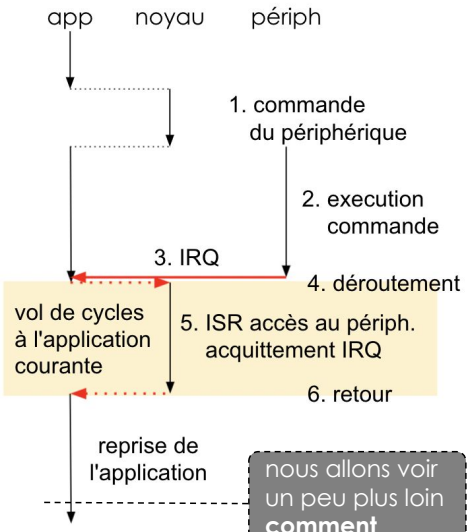
→ Cela dépend du contrôleur émetteur, mais il y a toujours un acquittement…

Gestionnaire d'interruptions du noyau

Le gestionnaire d'interruption est invoqué (3.) lorsqu'**une IRQ non masquée s'active**

Étapes de traitement (détaillées dans les slides suivants)

- Le MIPS se **Déroute** (4.) vers le noyau (*en 1 cycle*)
  `c0_EPC ← PC+4 ; c0_sr.EXL ← 1 ; c0_cause.XCODE ← 0`
  `PC ← 0x80000180`
- Le noyau analyse du champ `XCODE` de `c0_cause`
- Le noyau appelle le gestionnaire d'interruption
- Le noyau sauvegarde les registres temporaires
- Le noyau lit le numéro d'IRQ dans ICU_HIGHEST
- Le noyau **exécute l'ISR associée au numéro d'IRQ** (
  - il accède aux registres du périphérique
  - il acquitte l'IRQ (c.-à-d. baisse de la ligne d'interruption)
- Le noyau restaure les registres temporaires
- Le MIPS **Retourne** (6.) au programme interrompu avec l'instruction `eret : c0_sr.EXL ← 0 ; PC ← EPC`



ISR : Interrupt Service Routine

Un pilote de périphérique contient :

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

Les ISR (ou routines d'interruption) sont donc 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 WAIT à READY) de l'application qui était en attente de la terminaison de la commande*

pas dans cette version de l'OS

- 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

RAM

Sélection et appel de la bonne ISR

Ce schéma représente :

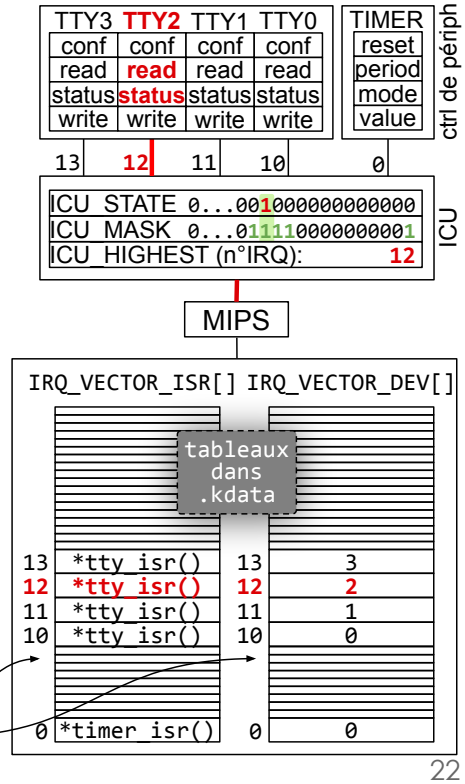
1. Les registres des contrôleurs de périphériques du SoC impliqués dans le traitement des IRQ
2. Le vecteur d'interruption permettant au noyau de savoir quelle ISR exécuter en fonction du numéro de l'IRQ active

Note 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 forcément `'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`

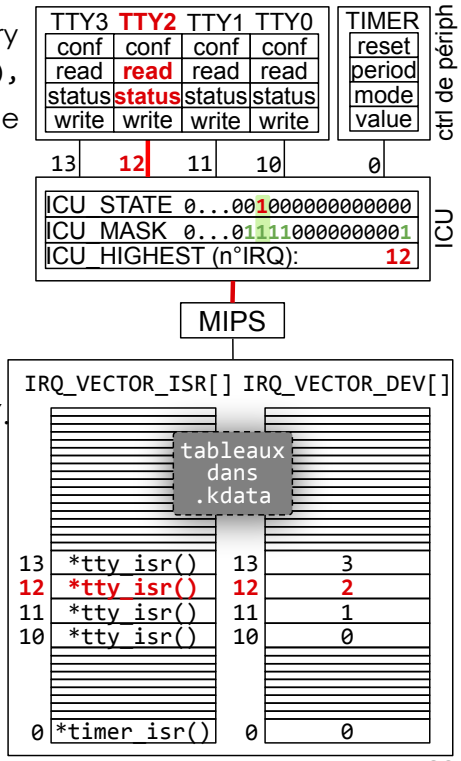
Nous allons voir maintenant l'usage du vecteur d'interruption et du vecteur de devices



RAM

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()`** 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[]` contiendrait un pointeur sur une structure de donnée propre au périphérique (structure « `device` »)

harch.c

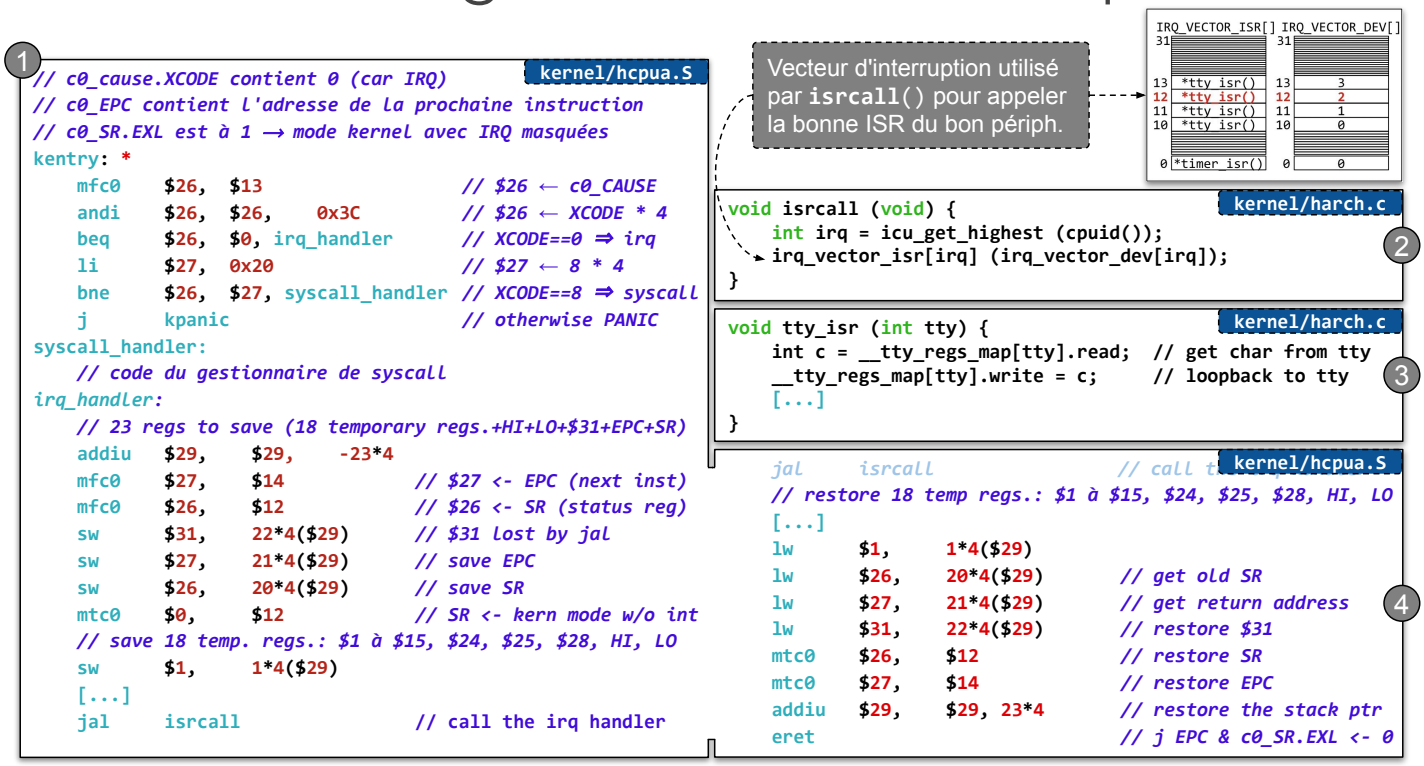
```
void arch_init (int tick) {
    timer_init (0, tick);
    icu_set_mask (0, 0);
    irq_vector_isr [0] = timer_isr;
    irq_vector_dev [0] = 0;

    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;
}
```

Code du gestionnaire d'interruption



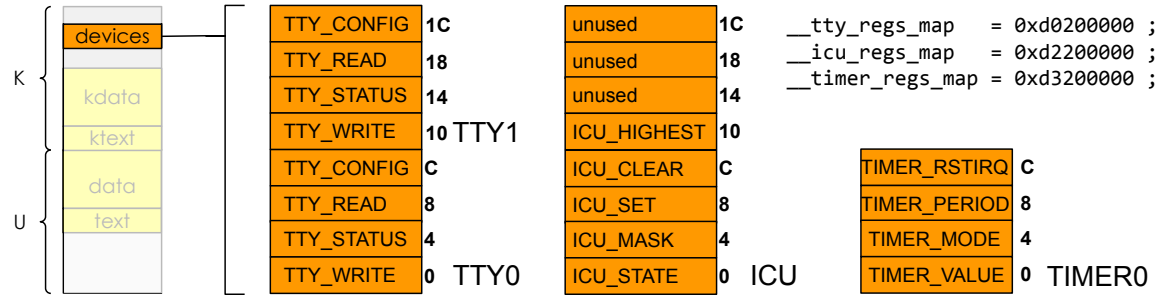
Ce qu'il faut retenir

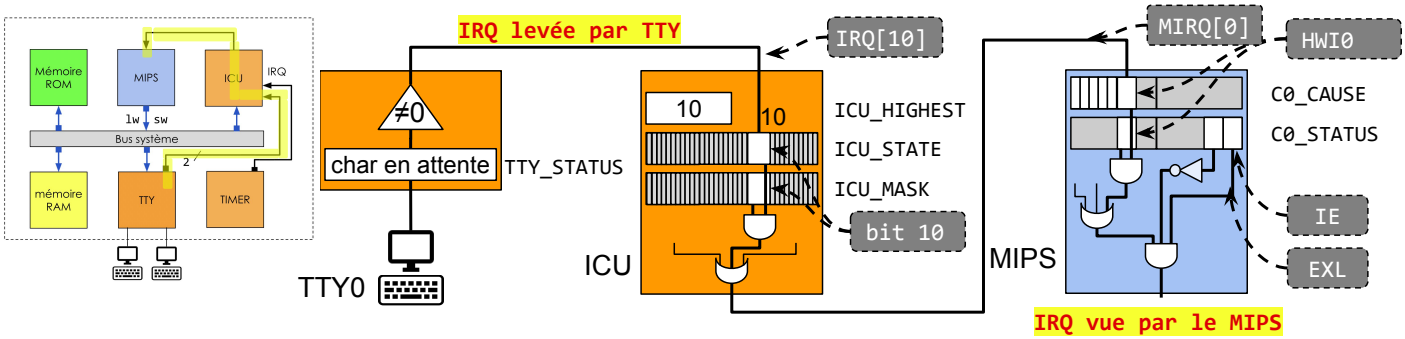
- Quand une IRQ non masquée est levée,
  elle provoque le déroutement du programme en cours vers le noyau.
- Le noyau voit que c'est une IRQ grâce au champ `c0_cause.XCODE`
  et il exécute alors le gestionnaire d'interruption.
- Grâce à l'ICU, le gestionnaire d'interruption sait quelle IRQ est levée.
- Le noyau utilise le numéro d'IRQ comme index pour lire le vecteur d'interruption contenant les adresses des ISR (routines d'interruption)
- Le noyau utilise le vecteur de *devices* pour connaître l'instance du périphérique.
- Le noyau exécute la bonne ISR en lui donnant en argument le numéro d'instance, l'ISR écrit dans les registres du périphérique et acquitte l'IRQ.
- Les ISR s'exécutent en "volant" des cycles aux applications interrompues.
- Les ISR ne sont pas interruptibles (c'est un choix simplificateur)
- La gestion d'une IRQ ne modifie aucun registre du programme interrompu, l'assembleur sauve les regs temporaires, le code C sauve les regs persistants.

Conclusion

- Résumé en schémas
- Quelles sont les étapes du TME
- et la suite du module

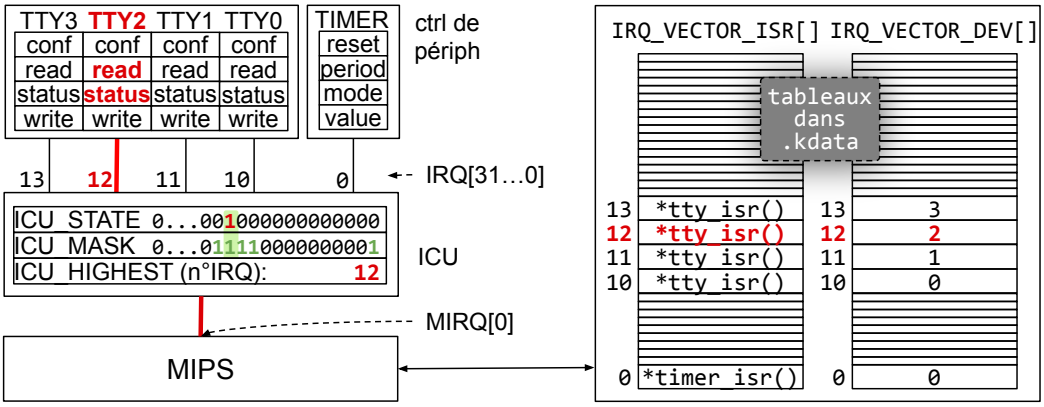
Devices & Niveau de masquage des IRQ





RAM

Sélection et appel de la bonne ISR



Etapes du TME

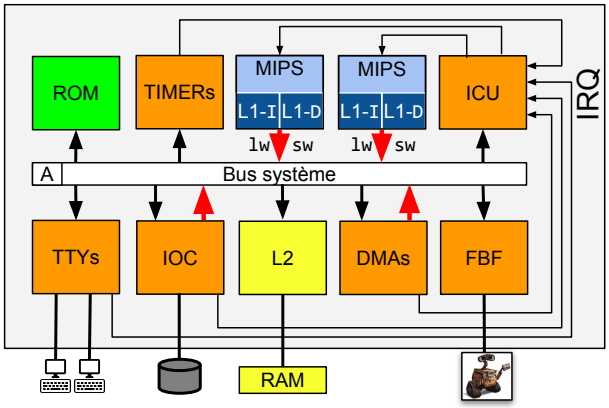
Au prochain TME, vous allez manipuler le gestionnaire d'interruption et le timer.

- L'idée va être d'exécuter un programme de jeu en mode user et d'utiliser le timer pour compter le nombre de cycles en parallèle.
- Nous allons faire deux traitements par le noyau dans l'ISR du timer
  1. Stopper définitivement le programme dès la première IRQ
  2. Stopper définitivement le programme après un certain nombre d'IRQ
- Enfin, vous allez estimer la durée en cycle du traitement d'une ISR en utilisant le dispositif de trace d'exécution proposé par le simulateur.

Il y a une suite... pour l'architecture

Dans l'UE `LU3IN031`,
concernant l'architecture, vous verrez

- L'architecture interne d'un MIPS,
  en particulier le séquencement des instructions
- la micro-architecture des opérateurs
  en vue de leur performance
- L'architecture d'un SoC avec plusieurs MIPS
  se partageant le même espace d'adressage
- L'architecture d'un cache de premier niveau
  et les problèmes de cohérence en multicores
- Le fonctionnement du contrôleur de disque
- Le fonctionnement d'un contrôleur graphique



Il y a une suite... pour l'OS

Dans l'UE `LU3IN031`,
concernant le système d'exploitation, vous verrez

- Une gestion de la mémoire dynamique,
  nécessaire pour créer des variables et les détruire
- Une API de gestion de listes chaînées pour construire
  des structures de données plus complexes
- Une gestion des états d'attente de threads et
  des listes d'attentes sur les ressources partagées
- Une gestion plus propre des pilotes de périphériques
- Les mécanismes de communications et
  de synchronisation des threads en mono-core
- Un système de gestion de fenêtre graphique que
  vous utiliserez pour réaliser un jeu 2D (genre Pong)
- et pleins d'autres petites choses pour la programmation
  système....