

[[Start](#)] [[Config](#)] [[MIPS User](#)] [[MIPS Kernel](#)] ? [[Cours 9](#)] [[Cours 10](#)] [[Cours 11](#)] ? [[?TME 9](#)] [[?TME 10](#)] [[TME 11](#)]

1. [A. Travaux dirigés](#)
 1. [Rappel de cours](#)
 2. [A. Travaux Dirigé](#)
2. [Questions de cours sur les interruptions](#)
3. [Exercices](#)
4. [TME sur les interruptions.](#)
 1. [Ajout de l'ISR timer](#)
 2. [Game Over](#)

Codes (tgz) ? [[gcc & simulateur](#)] [[TME 9](#)] [[TME 10](#)] [[TME 11](#)]

Gestionnaire d'interruptions et application multi-tâches en temps partagé

Cette page décrit la séance complète : partie TD et partie TP. Elle commence par la partie TD avec des questions ou des exercices à faire sur papier, réparties dans 4 sections. Certaines questions de sections différentes sont semblables, c'est normal, cela vous permet de réviser. Puis, dans la partie TP, il y a des questions sur le code avec quelques exercices de codage simples à écrire et à tester sur le prototype. La partie TP est découpée en 3 étapes. Pour chaque étape, nous donnons (1) une brève description avec une liste des objectifs principaux de l'étape, (2) une liste des fichiers avec un bref commentaire sur chaque fichier, (3) une liste de questions simples dont les réponses sont dans le code, le cours ou le TD et enfin (4) un petit exercice de codage.

IMPORTANT

Avant de faire cette séance, vous devez avoir lu les documents suivants :

- [Cours sur le gestionnaire d'interruption et les threads](#) : *obligatoire*
- [Document sur l'assembleur du MIPS et la convention d'appel des fonctions](#) : *recommandé, mais déjà lu*
- [Documentation sur le mode kernel du MIPS32](#) : *obligatoire*

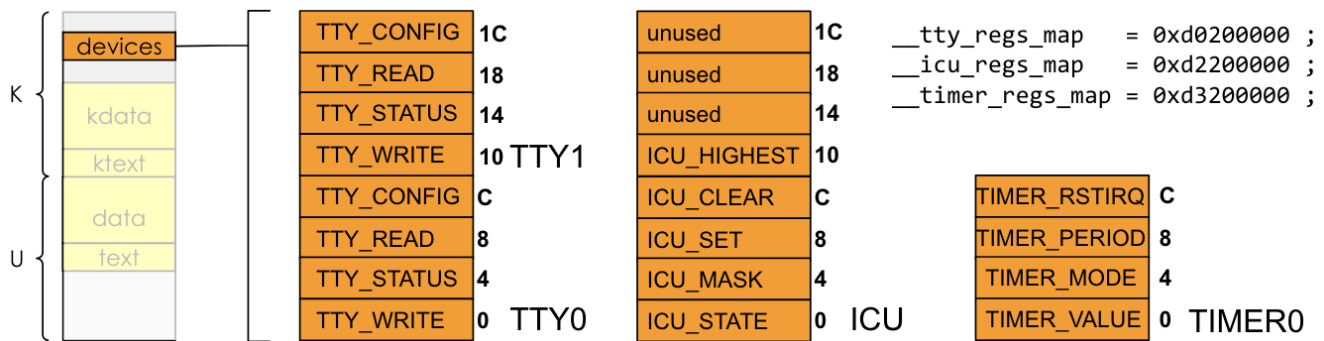
A. Travaux dirigés

Rappel de cours

Il est fortement recommandé de lire les transparents, toutefois, nous avons ajouté ci-après quelques rappels utiles pour répondre aux questions du TD.

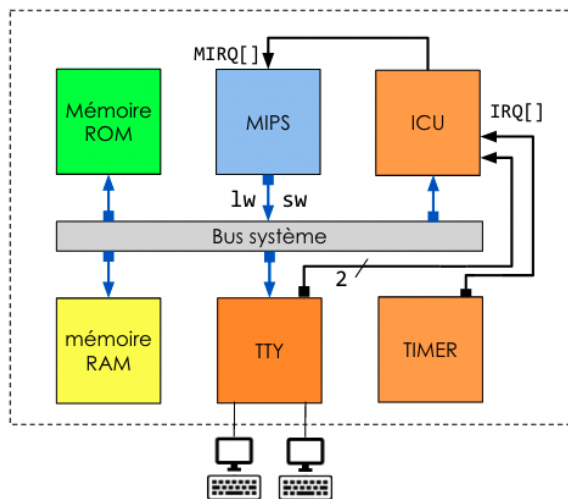
Dans cette séance, nous allons manipuler 3 contrôleurs de périphériques: Le TTY que vous connaissez déjà et deux autres, l'ICU et le TIMER. Ces trois contrôleurs s'utilisent grâce à des registres mappés (placés) dans l'espace d'adressage du MIPS. Les registres du TTY sont placés à partir de l'adresse 0xd0200000, ceux de l'ICU à partir de l'adresse 0xd1200000 et enfin ceux du TIMER à partir de l'adresse 0xd3200000. L'explication du rôle de ces registres est rappelée en partie dans ce texte et pour le détail, vous devez revoir le cours. Le choix des adresses de ces contrôleurs est fait par le créateur du matériel, elles ne peuvent pas être changées par le logiciel. Ces adresses sont données dans le fichier `ldscript` du kernel (`kernel.ld`) parce qu'elles ne sont utilisables que si le MIPS est

en mode kernel (adresses > 0x80000000).



Les IRQ (Interrupt ReQuest)s sont des signaux électriques à 2 états (ON/OFF ou Actif/Inactif ou encore Levé/Baissé). Les IRQ sont levés par les contrôleurs de périphériques pour prévenir d'un événement (fin de commande, arrivée d'une donnée, etc.). Les IRQs provoquent l'exécution d'ISR (Interrupt Service Routine) par le noyau. Les ISR sont des fonctions qui reçoivent en argument un identifiant du contrôleur de périphérique qui a levé l'IRQ. Une ISR doit faire deux choses, (1) accéder aux registres du contrôleur de périphérique concerné pour faire ce que le périphérique demande et (2) acquitter l'IRQ, c'est-à-dire demander au contrôleur de périphérique de baisser/désactiver son IRQ (puisque celle-ci a été traitée).

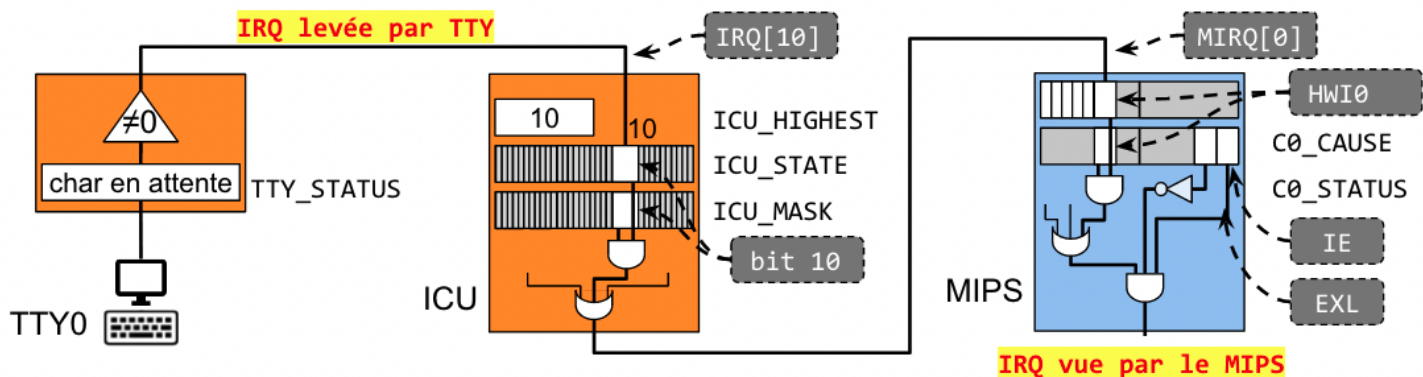
Les IRQ sont des signaux d'état qui doivent rester levés/activés tant qu'ils n'ont pas été acquittés par une ISR. Quand une IRQ se lève, la conséquence est que le programme en cours d'exécution sur le processeur recevant l'IRQ est interrompu et qu'il est dérouter vers le noyau pour que ce dernier exécute l'ISR prévue pour l'IRQ. Notez que ce n'est pas le processeur qui est interrompu, c'est bien le programme, car le processeur est seulement dérouter vers le noyau, mais il continue à travailler.



Sur le schéma de la plateforme des TP, on peut voir que ce sont seulement les composants TTY et TIMER qui peuvent lever des IRQ. Les IRQ de ces contrôleurs de périphériques sont envoyés au composant ICU qui va les combiner pour produire un unique signal IRQ pour le processeur.

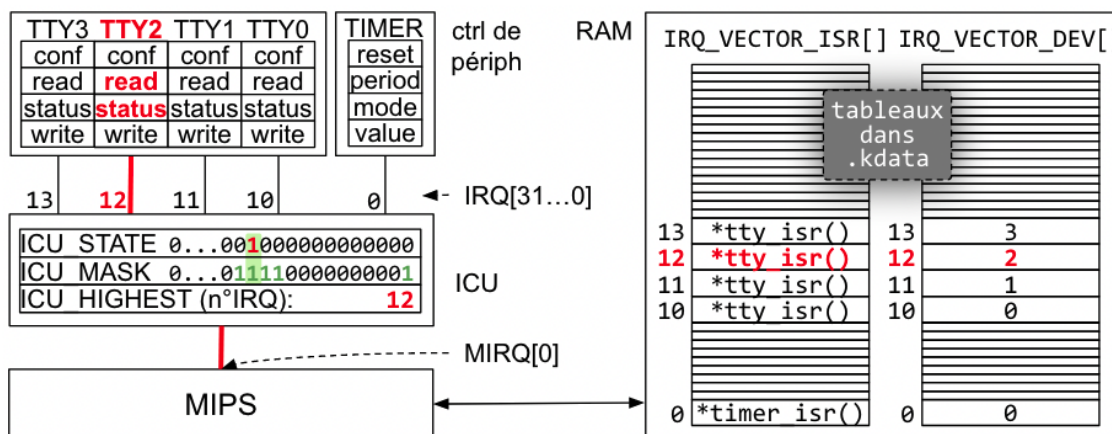
Une IRQ peut être masquée, c'est-à-dire que le processeur ne va pas interrompre le programme en cours. Le masquage peut être demandé à plusieurs endroits : dans le composant ICU et dans le processeur lui-même. Le masquage est demandé par le noyau, le plus souvent de manière temporaire, quand il doit exécuter un code critique

qui ne doit surtout pas être interrompu.



Sur le schéma ci-dessus, on voit que l'IRQ du TTY0 entre sur l'entrée n°10 de l'ICU, c'est un choix matériel qui n'est pas modifiable par logiciel. Son état est donc enregistré dans le bit n°10 du registre ICU_STATE. Il y a un AND avec le bit 10 du registre ICU_MASK. Si le bit 10 du registre ICU_MASK est à 0, alors la sortie du AND est 0 et l'IRQ est masquée (donc invisible pour le processeur). Le registre ICU_HIGHEST contient toujours le numéro de l'IRQ active la plus prioritaire, comme il n'y en a qu'une dans cet exemple, ICU_HIGHEST contient 10. L'IRQ de l'ICU entre sur l'entrée 0 des 6 IRQs possibles du MIPS et sa valeur s'inscrit dans le registre HWI0 du registre CO_CAUSE. Il y a un AND avec le bit HWI0 du registre CO_STATUS. Si le bit HWI0 du registre CO_STATUS est à 0, alors la sortie du AND est 0 et l'IRQ est aussi masquée. Enfin, il y a encore un AND qui permet de masquer globalement les IRQ avec le bit 0 de CO_STATUS (c'est le bit IE pour Interrupt Enable) et le NOT du bit 1 de CO_STATUS (c'est le bit EXL EXception Level).

Quand le signal IRQ vue par le MIPS s'active (passe à 1), c'est que l'IRQ levée par le contrôleur de périphérique doit être prise en charge. Le programme en cours est interrompu et dérouter vers `kentry` à l'adresse `0x80000180` et en même temps `CO_EPC ? PC+4`, `CO_CAUSE.XCODE ? 0`, `CO_STATUS.EXL ? 1`. Notez que le nom officiel de `CO_STATUS` est `CO_SR`, mais dans ce document, on utilise `CO_STATUS` pour plus de clarté.



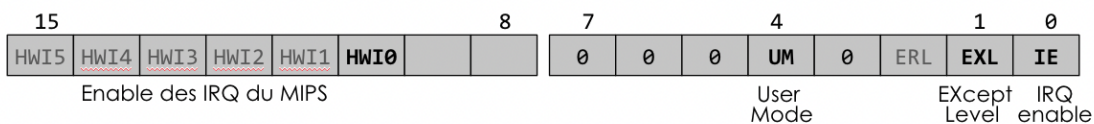
Dans le schéma précédent, à gauche c'est le matériel et à droite c'est un extrait de la RAM contenant les structures de données utilisées par le noyau.

- À gauche, on voit que les IRQ venant des contrôleurs de périphériques sont connectés aux entrées d'IRQ de l'ICU. Il y a 32 entrées possibles. Sur notre plateforme, par exemple l'IRQ du TTY2 est connectée à l'entrée 12 de l'ICU. Ce numéro d'entrée est le numéro qui identifie le contrôleur de périphérique. Notez que le

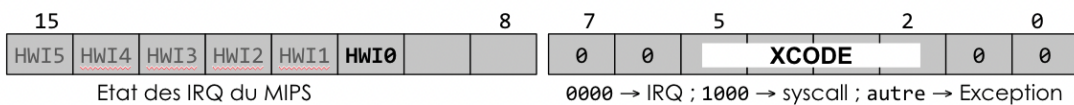
registre `ICU_MASK` est en lecture seul, c'est-à-dire qu'il ne peut pas être écrit directement. Pour modifier le contenu du registre `ICU_MASK`, il faut utiliser deux autres registres de l'ICU: `ICU_SET` et `ICU_CLEAR`. `ICU_SET` permet de mettre à 1 les bits de `ICU_MASK`, et `ICU_CLEAR` permet de les mettre à 0. Pour mettre à 1 le bit i du registre `ICU_MASK`, il faut écrire 1 dans le bit i du registre `ICU_SET`. Pour mettre à 0 le bit j du registre `ICU_MASK`, il faut aussi écrire 1, mais dans le bit j du registre `ICU_CLEAR`.

- À droite, il y a les deux tableaux que le noyau utilise pour connaître l'ISR à exécuter pour chaque numéro IRQ. Ce couple de tableaux se nomme **vecteur d'interruption**. Ici, il est composé des tableaux `IRQ_VECTOR_ISR[]` et `IRQ_VECTOR_DEV[]`. Le vecteur d'interruption est indexé par les numéros d'IRQ. Il contient deux informations: (1) Dans la case $n^o i$ du tableau `IRQ_VECTOR_ISR[]`, on trouve le pointeur sur la fonction ISR à appeler si l'IRQ $n^o i$ est levée, et (2) dans la case $n^o i$ du tableau `IRQ_VECTOR_DEV[]`, on trouve le numéro de l'instance du périphérique. Cette dernière information est nécessaire dans le cas des contrôleurs de périphérique multi-instances comme le TTY afin de savoir quel jeu de registres la fonction ISR doit utiliser. En d'autres termes, il y a une fonction ISR unique quelque-soit le numéro du TTY, l'adresse de cette fonction est placée dans les cases 10, 11, 12, et 13 du tableau `IRQ_VECTOR_ISR[]` (si on a 4 TTYs) et dans les cases 10, 11, 12, et 13 du tableau `IRQ_VECTOR_DEV[]`, on a 0, 1, 2 et 3 qui correspondent bien au numéro d'instance des TTYs.

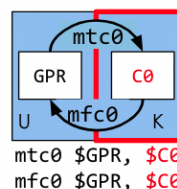
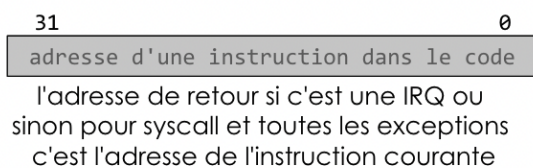
Le registre `c0_sr` (\$12) contient le mode d'exécution du MIPS et les autorisations d'IRQ



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



Le registre `c0_epc` (\$14)



Sur le schéma précédent, nous vous rappelons les 3 registres du coprocesseur système qui sont utilisés au moment de l'entrée dans le noyau, quelque soit la cause : syscall (vu la semaine dernière), interruption (TD de cette semaine) et exception (dans le cas de problèmes lors de l'exécution du programme comme la division par 0). On voit aussi que les seules instructions qui peuvent manipuler ces registres sont `mtc0` et `mfc0` pour, respectivement, les écrire et les lire.

Les bits `HWI0` des registres `C0_STATUS` (aussi nommé `c0_sr`) et `C0_CAUSE` contiennent respectivement le mask et le l'état de l'entrée $n^o 0$ d'interruption du MIPS. Les bits `UM`, `IE` et `EXL` sont liés au mode d'exécution du MIPS: `UM` est le bit de mode du MIPS (1=User Mode, 0=Kernel Mode), `IE` est le bit de masque général des interruptions (1=autorisées, 0=masquées) et enfin `EXL` est le bit que le MIPS met à 1 à l'entrée dans le noyau pour informer d'un niveau exceptionnel et dans ce cas les bits `UM` et `IE` ne sont plus significatifs, si `EXL` est à 1 alors le MIPS est en mode kernel, interruptions masquées.

A. Travaux Dirigé

La majorité des réponses aux questions ci-après sont dans le rappel du cours donné au début de cette page, c'est voulu.

Questions de cours sur les interruptions

1. A quelles adresses dans l'espace d'adressage sont placés les registres des 3 contrôleurs de périphériques de la plateforme et comment le kernel les connaît ?
2. Que signifie l'acronyme I.R.Q. ?
3. Une IRQ est un signal électrique, combien peut-il avoir d'états ?
4. Qu'est-ce qui provoque une IRQ ?
5. Les IRQ relient des composants source et des composants destinataires, quels sont ces composants ?
Donnez un exemple.
6. Que signifie masquer une IRQ ?
7. Quels composants peuvent masquer une IRQ ?
8. Est-ce qu'une application utilisateur peut demander le masquage d'une IRQ ?
9. Que signifie l'acronyme I.S.R. ?
10. Dans la plateforme des TPs, sur quelles entrées de l'ICU sont branchées les IRQ venant des TTYs et du TIMER ?
11. Quelle valeur mettre dans le registre `ICU_MASK` si on veut recevoir seulement les IRQ venant des 4 TTYs, dans le cas de la plateforme utilisée en TP ? Donnez le nombre en binaire et en hexadécimal.
12. L'écriture dans `ICU_MASK` n'est pas possible, comment modifier ce registre pour mettre à 1 le bit 0 ?
13. Dans quel mode est le processeur quand il traite une IRQ ?
14. Que fait le processeur lorsqu'il reçoit une IRQ masquée ?
15. Que signifie acquitter une IRQ et qui le demande à qui ?
16. Est-ce qu'une IRQ peut se désactiver sans intervention du processeur ?
17. Est-ce qu'une IRQ peut ne pas être attendue par le noyau ?
18. Quelle est la valeur du champ `XCODE` du registre `c0_cause` à l'entrée dans le noyau en cas d'interruption ?
19. Quelle est la valeur écrite dans le registre `c0_EPC` à l'entrée dans le noyau en cas d'interruption ?
20. Que se passe-t-il dans le registre `c0_sr` à l'entrée dans le noyau en cas d'interruption et quelle est la conséquence ?
21. La routine `kentry` (entrée du kernel à l'adresse `0x80000180`) appelle le gestionnaire d'interruption quand le MIPS reçoit une IRQ non masquée, que fait ce gestionnaire d'interruption ?
22. À l'entrée dans le noyau, `kentry` analyse le champ `XCODE` du registre de `c0_cause` et si c'est 0 alors il saute au code donné ci-après (ce n'est pas exactement le code que vous pouvez voir dans les fichiers sources pour que ce soit plus facile à comprendre).

```
cause_irq:
    addiu    $29,    $29,    -23*4        // 23 registers to save (18 tmp regs+HI+LO+$31+EP
    mfc0     $27,    $14                    // $27 <- EPC (addr of syscall instruction)
    mfc0     $26,    $12                    // $26 <- SR (status register)
    sw       $31,    22*4($29)              // $31 because, it is lost by jal irq_handler
    sw       $27,    21*4($29)              // save EPC (return address of IRQ)
    sw       $26,    20*4($29)              // save SR (status register)
    mtc0     $0,     $12                    // SR <- kernel-mode without INT (UM=0 ERL=0 EXL=
    sw       $1,     1*4($29)              // save all temporary registers including HI and
    sw       $2,     2*4($29)
    [etc.]

    jal      irq_handler                    // call the irq handler fonction écrite en C
```

```

lw      $1,      1*4($29)           // restore all temporary registers including HI a
lw      $2,      2*4($29)
[etc.]
lw      $26,     20*4($29)          // get old SR
lw      $27,     21*4($29)          // get return address of syscall
lw      $31,     22*4($29)          // restore $31
mtc0    $26,     $12                // restore SR
mtc0    $27,     $14                // restore EPC
addiu   $29,     $29, 23*4          // restore the stack pointer
eret                                         // jr C0_EPC AND C0_SR.EXL <= 0

```

Pourquoi, ne pas sauver les registres persistants ?

23. La fonction `irq_handler()` a pour mission d'appeler la bonne ISR. Dans le code qui suit (extrait du fichier `kernel/harch.c`), on voit d'abord la déclaration de la structure qui décrit les registres présents dans l'ICU. En fait c'est un tableau de structure parce qu'il y a autant d'instances d'ICU que de processeurs (donné par `NCPUS`), ici, il y a un seul processeur MIPS, donc `NCPUS=1`.

```

struct icu_s {
    int state;           // state of all IRQ signals
    int mask;           // IRQ mask to chose what we need for this ICU
    int set;            // IRQ set --> enable specific IRQs for this ICU
    int clear;          // IRQ clear --> disable specific IRQs for this ICU
    int highest;        // highest pritority IRQ number for this ICU
    int unused[3];      // these 3 registers are not used
};

extern volatile struct icu_s __icu_regs_map[NCPUS];

static int icu_get_highest (int icu) {
    return __icu_regs_map[icu].highest;
}

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

void irq_handler (void) {
    int irq = icu_get_highest (cpuid());
    irq_vector_isr[irq] (irq_vector_dev[irq]);
}

```

La déclaration `extern volatile struct icu_s __icu_regs_map[NCPUS];` informe le compilateur que le symbole `__icu_regs_map` est défini ailleurs et que c'est un tableau de structures de type `struct icu_s`. Ainsi, `gcc` sait comment utiliser cette variable `__icu_regs_map`.

Rappelez dans quel fichier est défini `__icu_regs_map` ?

Que font les fonctions `icu_get_highest()`, `icu_set_mask()` et `irq_handler()` ?

Comment s'appelle le couple de tableaux `irq_vector_isr[irq]` et `irq_vector_dev[irq]` ?

Combien ont-il de cases ?

24. Si `ICU_HIGHEST` contient 10 (dans le cas de notre plateforme) que doit faire la fonction `irq_handler()` ?
25. Que fait la fonction `icu_set_mask (int icu, int irq) ?`

Exercices

1. La configuration des périphériques et des interruptions est faite dans la fonction `arch_init()` appelée par `kinit()`.

Comment configurer l'ICU pour démasquer l'IRQ connectée sur son entrée n°0 ?

1. Supposons que l'on ait défini les registres du TIMER de la façon suivante :

TME sur les interruptions.

Ajout de l'ISR timer

Game Over

Dans le premier TME, vous avez réalisé un petit jeu dans lequel vous deviez deviner un nombre tiré au hasard. Ce jeu avait été mis dans `kinit` parce qu'à ce moment, il n'y avait pas encore d'application utilisateur. Nous vous proposons de mettre le jeu dans l'application `user` et de limiter le temps pendant lequel vous pouvez jouer.

Modifier l'ISR du timer pour afficher "Game Over"

[...]