

## INDEX

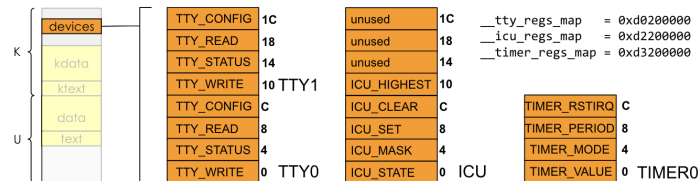
DOC → [Config] [MIPS U] [MIPS K] [markdown] [CR.md]  
 COURS → [1] (+code) (+outils) [2] [3] [4] [5] [6] [7] [8] [9]  
 TME → [1] [2] [3] [4] [5] [6] [7] [8] [9]  
 CODE → [gcc + soc] [1] [2] [3] [4] [5] [6] [7] [8] [9]

A. Questions  
 B. Travaux pratiques  
 1. La plateforme  
 2. Game over simple  
 3. Game over avec décompteur  
 4. Évaluation de la durée d'une ISR

## 2 - Gestionnaire d'interruptions

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

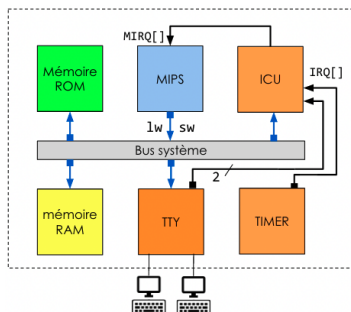
Dans cette séance, nous allons manipuler 3 contrôleurs de périphériques: Le TTY que vous connaissez déjà et deux autres, l'ICU et le TIMER. Ces trois contrôleurs s'utilisent grâce à des registres mappés (placés) dans l'espace d'adressage du MIPS. Les registres du TTY sont placés à partir de l'adresse `0xd0200000`, ceux de l'ICU à partir de l'adresse `0xd2200000` et enfin ceux du TIMER à partir de l'adresse `0xd3200000`. Le rôle de ces registres est rappelé en partie dans ce texte et pour plus de détails, 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`).



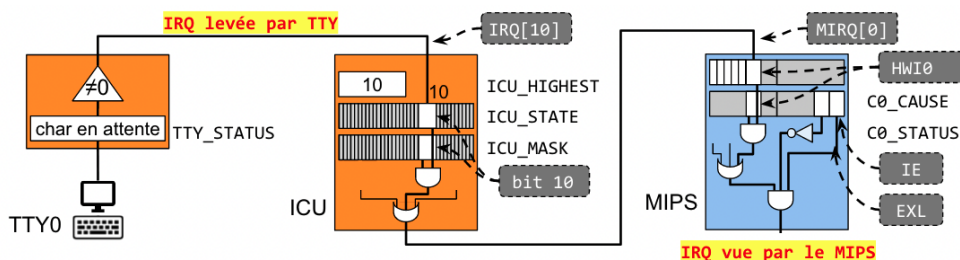
Les IRQ (Interrupt ReQuest)s sont des signaux électriques à 2 états (ON/OFF ou Actif/Inactif ou encore Levé/Baissé). Les IRQ sont levés par les contrôleurs de périphériques pour prévenir d'un événement (fin de commande, arrivée d'une donnée, etc.). Les IRQs provoquent l'exécution d'ISR (Interrupt Service 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). La demande d'acquiescement est spécifique à chaque contrôleur de périphérique. Pour le TTY, il faut lire le registre `TTY_READ`. Pour le TIMER, il faut écrire dans le registre `TIMER_RSTIRQ`.

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 dérivé 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 dérivé vers le noyau, mais il continue à travailler.

Sur le schéma de la plateforme des TP, on peut voir que seuls les composants TTY et TIMER 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.

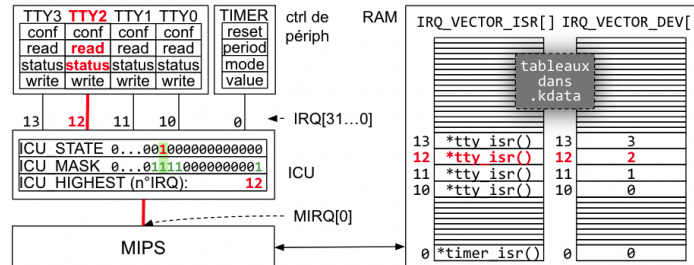


Sur le schéma ci-dessus, on voit que l'IRQ du TTY0 est reliée à l'entrée n° 10 de l'ICU, c'est un choix matériel qui n'est pas modifiable par logiciel. Son état est donc enregistré dans le bit n°10 du registre `ICU_STATE`. Il y a un `AND` avec le bit 10 du registre `ICU_MASK`. Si le bit 10 du registre `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 prioritaire, pour cette ICU, est l'IRQ active dont le numéro est le plus petit). L'IRQ de l'ICU est reliée à l'entrée 0 des 6 IRQs possibles du MIPS et sa valeur s'inscrit dans le registre `HWIO0` du registre `C0_CAUSE`. Il y a un `AND` avec le bit `HWIO0` du registre `C0_STATUS`. Si le bit `HWIO0` du registre `C0_STATUS` est à 0, alors la sortie du `AND` est 0 et l'IRQ est aussi masquée. Enfin, il y a un dernier `AND` avec le bit 0 de `C0_STATUS` (correspondant au bit `IE` pour Interrupt Enable) qui permet de masquer globalement les IRQ et avec le `NOT` du bit 1 de `C0_STATUS` (correspondant au bit `EXL` Exception Level).

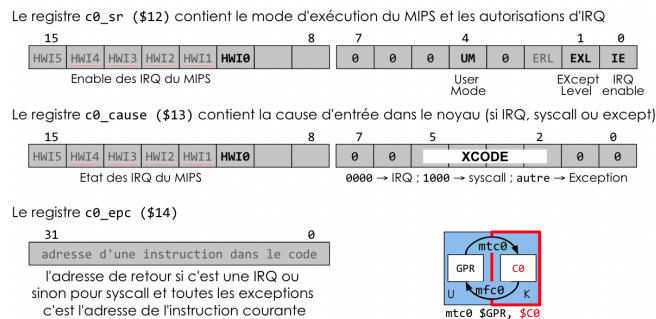
Quand le signal IRQ vue par le MIPS s'active (passe à 1), c'est que l'IRQ levée par le contrôleur de périphérique doit être prise en charge. Le programme en cours d'exécution est interrompu et dérivé vers `kentry` à l'adresse `0x80000180` et en même temps `C0_EPC ← PC+4`, `C0_CAUSE.XCODE ← 0`, `C0_STATUS.EXL ← 1`. Notez que le nom officiel de `C0_STATUS` est `C0_SR`, mais dans ce document, on utilise `C0_STATUS` pour plus de clarté.

Dans le schéma ci-après, 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 pour la gestion des IRQ.

- 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 d'IRQ. Ce couple de tableaux se nomme **vecteur d'interruption** et comme il y a 32 entrées d'IRQ dans l'ICU, ces tableaux ont 32 cases chacun. Ici, le vecteur d'interruption 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° `i` du tableau `IRQ_VECTOR_ISR[]`, on trouve le pointeur sur la fonction ISR à appeler si l'IRQ n° `i` est levée, et (2) dans la case n° `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 effet, il y a une fonction ISR unique à exécuter quel que 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 les valeurs 0, 1, 2 et 3 qui correspondent bien au numéro d'instance des TTYs.



Enfin, nous rappelons les 3 registres du coprocesseur système (`c0`) qui sont utilisés au moment de l'entrée dans le noyau, quelle que 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 rappelle aussi que les seules instructions qui peuvent manipuler ces registres sont `mtc0` et `mfc0` pour, respectivement, les écrire et les lire.



Les bits `HWIO` des registres `c0_status` (aussi nommé `c0_sr`) et `c0_cause` contiennent respectivement le mask et l'état de l'entrée n° 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, et les interruptions sont masquées.

## A. Questions

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

- À quoi servent les interruptions ?
  - Les interruptions sont un mécanisme permettant de traiter au plus vite un événement matériel ou une demande d'un autre processeur.
- Une interruption en informatique est à la fois une suspension temporaire d'un programme et un signal électrique. Comment s'appelle le signal d'interruption et comment s'appelle le code permettant de la traiter ?
  - Le signal d'interruption se nomme IRQ comme Interrupt ReQuest?.
  - Le code pour traiter une IRQ est une ISR comme Interrupt Service Routine
- Est-ce que tous les composants génèrent des signaux d'interruption ? Si la réponse est non, donnez un exemple ?
  - Non, les mémoire ne produisent pas d'interruption et, ici, le *frame buffer* (la mémoire vidéo) non plus.
- Est-ce qu'un composant peut produire plusieurs signaux d'interruption ?
  - Oui, le TTY gère plusieurs terminaux, donc plusieurs clavier, et chacun produit une IRQ.
- Est-ce qu'une application utilisateur sait quand elle va être interrompue ?
  - Non, elle sera interrompue, mais elle ne peut pas savoir quand.
- Est-ce qu'une application utilisateur sait quand elle a été interrompue ?
  - Non plus, le noyau lui a volé du temps d'exécution, mais elle ne sait pas quand...
- Que signifie IPI et à quoi ça sert ? (Il n'y a pas de slides dans le cours sur ça...)
  - IPI signifie Inter-Processor Interrupt, c'est un mécanisme permettant à un programme d'interrompre un autre programme s'exécutant sur un autre processeur. Il faut un composant matériel spécifique contenant des registres dont l'écriture provoque des IRQ.
- Est-ce qu'un programme utilisateur peut interdire les interruptions en général ?
  - Non, il faudrait écrire dans un registre de l'ICU ou dans le registre `c0_sr` qui appartiennent au noyau.

9. Est-ce que le noyau du système d'exploitation peut interdire les interruptions en général ?
  - Oui, bien sûr, il le fait soit définitivement quand un périphérique n'est pas utilisé, ou temporairement pour les sections critiques.
10. Quand un signal d'interruption s'active est-ce que le noyau sait toujours quoi faire ?
  - Oui, c'est défini par le vecteur d'interruption, à chaque IRQ, il y a une ISR.
11. Que signifie l'expression «vol de cycles» ?
  - C'est une expression pour dire que l'application en cours a été interrompu le noyau pour exécuter une ISR.
12. Est-ce l'application qui a provoqué l'activation d'un signal d'interruption qui est volée ?
  - Non, s'il y a plusieurs applications en cours s'exécutant en temps partagé (on le verra au prochain cours), les IRQs arrivent n'importe quand, donc potentiellement lorsque le processeur s'occupe d'une autre application. C'est même le cas le plus fréquent.
13. Pour le composant TTY, à quel moment produit-il un signal d'interruption ?
  - Quand on appuie sur une touche
14. Pour le composant TTY, comment fait-on pour acquitter une d'interruption ?
  - En lisant, registre `TTY_READ`.
15. Si plusieurs caractères ASCII sont en attente dans d'être lus dans un TTY, quelle conséquence cela a-t-il sur le signal d'interruption ?
  - Il reste actif tant qu'un caractère est en attente.
16. À quoi sert le composant `TIMER` ?
  - Il sert à activer des interruptions périodiquement. On verra son usage pour l'exécution des programme en temps partagé.
17. Comment fait-on pour le configurer ?
  - Il faut écrire dans les registres `TIMER_PERIOD` et `TIMER_MODE`, le premier pour définir la période, le second pour démarrer le timer et demander à ce qu'il lève un IRQ à chaque période.
18. Comment fait-on pour acquitter une interruption pour le composant `TIMER` ?
  - Il faut écrire n'importe quelle valeur dans le registre `TIMER_RESETIRQ`
19. Est-ce que le registre `TIMER_VALUE` peut activer (on dit aussi lever) un signal d'interruption ?
  - Non, c'est seulement un compteur de cycles.
20. À 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 ?
  - `tty_regs_map = 0xd0200000 ;`
  - `icu_regs_map = 0xd2200000 ;`
  - `timer_regs_map = 0xd3200000 ;`
  - Ces adresses sont définies dans le ldscript du kernel `kernel.ld`, elles doivent être déclarées `extern` dans les codes C qui les utilisent.
21. Que signifie l'acronyme I.R.Q. ?
  - Interrupt ReQuest ou, en français, requête d'interruption
22. Une IRQ est un signal électrique, combien peut-il avoir d'états ?
  - C'est un signal à 2 états, c'est binaire. Il y a l'état `ON` (on dit aussi levé ou actif) pour dire que l'interruption est demandée et il y a l'état `OFF` (on dit aussi baissé ou inactif) pour dire que l'interruption n'est pas demandée.
23. Qu'est-ce qui provoque une IRQ ?
  - C'est un événement matériel sur le contrôleur de périphérique, comme la fin d'une commande ou l'arrivée d'une donnée.
24. Les IRQ relient des composants sources et des composants destinataires, quels sont ces composants ? Donnez un exemple.
  - Les composants sources sont les contrôleurs de périphériques par exemple le `TTY` et les composants destinataires sont les processeurs (ici le MIPS).
25. Que signifie masquer une IRQ ?
  - Cela signifie que l'on bloque le signal entre sa source et sa destination. Si une IRQ est levée par un contrôleur de périphérique et que l'on masque cette IRQ, alors l'IRQ est toujours levée à sa source, mais le MIPS destinataire ne le voit pas. L'information, le signal, a été masquée sur le chemin entre la source et la destination. Cette IRQ devient invisible pour le MIPS.
26. Quels composants peuvent masquer une IRQ ?
  - Ici, c'est le composant ICU et le MIPS lui-même.
27. Est-ce qu'une application utilisateur peut demander le masquage d'une IRQ ?
  - Non, puisque pour masquer une interruption, il faut écrire dans les registres de l'ICU ou dans les registres système du processeur. Or, les registres de configuration de l'ICU sont mappés dans la partie de l'espace d'adressage inaccessible en mode user et que les instructions `mfc0` et `mtc0` sont interdites en mode user, leur usage provoque une exception de type violation de privilège.
28. Que signifie l'acronyme I.S.R. ?
  - Interrupt Service Routine ou, en français, routine d'interruption. En fait, c'est une fonction C normale.
29. Dans la plateforme des TP, sur quelles entrées de l'ICU sont branchées les IRQ venant des TTYs et du `TIMER` ?
  - Les 4 IRQ de TTYs sont branchées sur les entrées `10`, `11`, `12` et `13` de l'ICU et l'IRQ du `TIMER` est sur l'entrée `0`.
30. Quelle valeur faut-il avoir 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.
  - Il faut que les bits `10`, `11`, `12` et `13` de `ICU_MASK` soit à 1 donc `0b00000000.00000000.00111100.00000000` donc `0x00003C00`.
31. L'écriture dans `ICU_MASK` n'est pas possible, comment modifier ce registre pour mettre à 1 le bit `0` ?
  - Il faut écrire `1` dans le bit `0` de `ICU_SET`.

32. Sur une plateforme (autre que celle des TP) sur laquelle on aurait un TTY0 sur l'entrée 5, un TIMER sur l'entrée 2, et un autre TTY1 sur l'entrée 14. Que doit-on faire pour que seuls le TTY1 et le TIMER soient démasqués et que TTY0 soit masqué ?  
Si les 3 IRQ se lèvent au même cycle, quelles seront les valeurs des registres `ICU_STATE`, `ICU_MASK` et `ICU_HIGHEST` ?
- on doit écrire `1` dans les bits 2 et 14 du registre `ICU_SET` donc `0b00000000.00000000.01000000.00000100 = 0x00004004`
  - on doit écrire `1` dans le bit 5 du registre `ICU_CLEAR` donc `0b00000000.00000000.00000000.00100000 = 0x00000020` pour être sûr que le bit 5 de `ICU_MASK` soit à 0. (Au reset, tous les bits de `ICU_MASK` sont à 0, mais là on ne sait pas si c'est juste après le reset)
  - Si les 3 IRQ s'activent alors `ICU_STATE` ← `0x00000000.00000000.01000000.00100100 = 0x00004024`  
on ne sait pas ce qu'il y a dans `ICU_MASK`, sauf pour les bits 2, 5 et 14 mais on sait qu'il ne change pas de valeur et `ICU_HIGHEST` ← `2` (le plus petit numéro d'IRQ).
33. Dans quel mode est le processeur quand il traite une IRQ ?
- Le MIPS est dans le mode kernel.
34. À quel moment doit-on initialiser le vecteur d'interruption ?
- Ici, c'est demandé par la fonction `kinit()`.
35. En quoi consiste la liaison des interruptions (*interrupt binding* en anglais) ?
- C'est le fait de lier une IRQ de périphérique et une ISR, c'est fait par le vecteur d'interruption.
36. Comment le noyau sait-il que la cause de son invocation est une interruption ?
- Dans `kentry`, c'est en analysant le champ `XCODE` du registre `c0_cause`.
37. Quelle instruction permet de sortir du noyau pour revenir dans le code interrompu ? et que fait-elle précisément ?
- c'est l'instruction `eret` qui fait : `c0_sr.EXL` ← `0` et `PC` ← `EPC`
38. Rappeler la différence entre un registre temporaire et un registre persistant.
- C'est lors d'un appel de fonction, les registres temporaires peuvent perdre leur valeur parce que la fonction appelée peut les utiliser sans être obligée de restaurer leur valeur.
  - A contrario, les registres persistants conservent leur valeur. C'est-à-dire que si une fonction appelée veut utiliser un registre persistant, elle doit sauver sa valeur à l'entrée de la fonction, pour la restaurer avant de revenir à la fonction appelante.
39. Pour qu'une IRQ soit effectivement prise en compte, il faut que le périphérique la lève et qu'elle ne soit pas masquée. Il y a plusieurs endroits où on peut masquer une IRQ, lesquels ?
- On peut parfois demander au composant qui produit l'IRQ de ne pas la produire, puis il y a l'ICU (en configurant le registre `ICU_MASK`) et le processeur lui-même (en configurant le registre `c0_sr`).
40. Que fait le processeur lorsqu'il reçoit une IRQ masquée ?
- Il ne fait rien puisqu'il ne la voit pas.
41. Que signifie acquitter une IRQ ?
- Cela signifie demander au contrôleur de périphérique concerné de baisser (désactiver) le signal IRQ.
42. Qui demande l'acquiescement à qui ?
- C'est l'ISR qui fait cette demande en accédant aux registres du contrôleur de périphérique. -.
43. Comment demande-t-on l'acquiescement ?
- poliment :-)
  - Chaque périphérique a sa propre méthode d'acquiescement, il faut lire la documentation de chaque composant pour le savoir. Pour le TTY, l'acquiescement est fait en lisant le registre `TTY_READ`, pour le TIMER, l'acquiescement est fait en écrivant n'importe quelle valeur dans le registre `TIMER_RSTIRQ`.
44. Est-ce qu'une IRQ peut se désactiver sans intervention du processeur ?
- Non, quand une IRQ est levée, elle ne peut être désactivée que par le code de l'ISR concernée. Le contrôleur de périphérique n'a pas le droit de la désactiver tout seul.
45. Est-ce qu'une IRQ peut ne pas être attendue par le noyau ?
- Non, le noyau ne doit pas être surpris par une IRQ, il doit avoir une ISR prévue. S'il doit traiter une IRQ non prévue, il affiche un message d'erreur, puis il masque cette IRQ dans l'ICU. Cela ne devrait jamais arriver.
46. Quelle est la valeur du champ `XCODE` du registre `c0_cause` à l'entrée dans le noyau en cas d'interruption ?
- Il y a `0`, et pour rappel `c0_cause.XCODE = 8` pour un syscall, les autres valeurs sont des numéros d'exception (division par 0, violation de privilège, etc.)
47. Quelle est la valeur écrite dans le registre `c0_EPC` à l'entrée dans le noyau en cas d'interruption ?
- C'est l'adresse de retour dans le programme interrompu. Quand le processeur reçoit une IRQ alors qu'il est en train d'exécuter l'instruction `i` à l'adresse `PC` (Program Counter), alors le MIPS termine l'exécution de l'instruction `i`, puis il enregistre `PC+4` (adresse de l'instruction qui suit `i`) dans `c0_EPC` et il saute à l'adresse `0x80000180`.
  - Question de Karine : et si `i` est un saut (ou l'instruction dans le delayed slot d'un saut pris ?)
48. Que se passe-t-il dans le registre `c0_status` à l'entrée dans le noyau en cas d'interruption et quelle est la conséquence ?
- Le bit `EXL` passe à 1 et la conséquence est que le MIPS passe en mode kernel, toutes les interruptions masquées quelles que soient les valeurs de `c0_status.UM` et `c0_status.IE` (respectivement le mode d'exécution et le masque d'interruption général).
49. 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 ?
- Le gestionnaire d'interruption doit déterminer le numéro de l'IRQ en lisant dans le registre `ICU_HIGHEST` de l'ICU et il doit appeler la fonction ISR trouvée dans le tableau `IRQ_VECTOR_ISR[]` du vecteur d'interruption dans la case du numéro de l'IRQ, en lui donnant en argument le numéro du périphérique (DEVICE) trouvé dans le tableau `IRQ_VECTOR_DEV[]` dans la case du numéro de l'IRQ.
50. 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+EPC+SR)
    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=0 IE=0)
sw      $1,     1*4($29)           // save all temporary registers including HI and LO
sw      $2,     2*4($29)
[etc. pour les autres sauvegardes des registres temporaires]

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

lw      $1,     1*4($29)           // restore all temporary registers including HI and LO
lw      $2,     2*4($29)
[etc. pour les autres restaurations des registres temporaires]
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 ?

- On doit sauver les registres temporaires parce que l'IRQ peut interrompre le programme n'importe quand et qu'il faudra revenir à l'application interrompue dans le même état donc sans perte d'information dans les registres. On ne sauve pas les registres persistants parce que ce sera fait dans la fonction `irq_handler()`, si c'est nécessaire.

51. 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 structures 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 priority 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, le compilateur `gcc` sait comment utiliser la variable `__icu_regs_map`.

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 ?

- Ce symbole est défini dans le fichier ldscript du kernel `kernel/kernel.ld`
- `icu_get_highest()` lit le registre `ICU_HIGHEST` de l'ICU et rend donc le numéro de l'IRQ la plus prioritaire.
- `icu_set_mask()` met 1 dans le bit n° `irq` du registre `ICU_SET` de l'ICU n° `icu` (ici `icu` est à 0 parce qu'il faut une ICU par MIPS et qu'il n'y a qu'un seul MIPS). Cela a pour effet de mettre à 1 dans le bit n° `irq` du registre `ICU_MASK`.
- `irq_handler()` va chercher dans l'ICU le numéro de l'IRQ la plus prioritaire et la copie dans la variable `irq` (cette notion de priorité n'a de sens que dans le cas où au moins deux IRQ sont actives en même temps). `irq_handler()` appelle la fonction ISR qui est dans la case n° `irq` du tableau `irq_vector_isr[]` et lui donne en argument le numéro d'instance qui est dans la case n° `irq` du tableau `irq_vector_dev[]`.
- Les deux tableaux constituent le vecteur d'interruption et ils ont autant de cases que l'ICU prend d'IRQ, c.-à-d. 32.

52. Si `ICU_HIGHEST` contient 10 (dans le cas de notre plateforme) que doit faire la fonction `irq_handler()` ?

- Si `ICU_HIGHEST` contient 10, c'est que c'est une IRQ du TTY0 et donc il faut appeler l'ISR du TTY en lui passant 0 en argument.

53. Que fait la fonction `icu_set_mask(int icu, int irq)` ?

- Elle met à 1 le bit `irq` du registre `ICU_MASK` de l'ICU n° `icu` (ici c'est nécessairement 0) puisqu'il n'y a qu'un seul MIPS donc une seule ICU).

54. Les registres du TIMER sont définis dans le code du noyau de la façon suivante :

```

struct timer_s {
    int value;           // timer's counter : +1 each cycle, can be written
    int mode;            // timer's mode : bit 0 = ON/OFF ; bit 1 = IRQ enable
    int period;          // timer's period between two IRQ
    int resetirq;        // address to acknowledge the timer's IRQ
};
extern volatile struct timer_s __timer_regs_map[NCPUS];

```

Écrivez le code de la fonction `static void timer_init(int timer, int tick)` qui initialise la période du timer n° `timer` avec l'entier nommé `tick` et active les IRQ si la période donnée est non nulle.

```

static void timer_init(int timer, int tick)
{
    __timer_regs_map[timer].period = tick;    // next period
}

```



```
__timer_regs_map[timer].mode = (tick)?3:0; // timer ON with IRQ only if (tick != 0)
}
```

55. La configuration des périphériques et des interruptions est faite dans la fonction `arch_init()` appelée par `kinit()`. Écrivez les instructions C permettant d'ajouter le TIMER dans le noyau avec un tick de 1000000 (1 million de cycles). Il faut (1) initialiser le timer ; (2) démasquer l'IRQ venant du timer dans l'ICU, elle connectée sur son entrée n°0 ; (3) initialiser le vecteur d'interruption avec la fonction `timer_isr` pour ce timer 0.

- Il faut une séquence de 4 instructions :

```
int tick = 1000000;
timer_init (0, tick);           // sets period of timer n'0 (thus for CPU n'0) and starts it
icu_set_mask (0, 0);           // [CPU n'0].IRQ <-- ICU.PIN[0] <- Interrupt signal timer n'0
irq_vector_isr [0] = timer_isr; // tell the kernel which isr to exec for ICU.PIN n'0
irq_vector_dev [0] = 0;         // device instance attached to ICU.PIN n'0
```

## B. Travaux pratiques

### 1. La plateforme

Le but de ce TP est d'analyser, de modifier et d'utiliser le gestionnaire d'interruption.

La plateforme que nous allons utiliser contient :

- un processeur
- une mémoire multisection pour le code et les données du noyau et de l'utilisateur.
- une ROM pour le boot
- un contrôleur MULTITTY
- un timer
- une icu

Sur cette plateforme, les composants produisant des IRQ sont le timer et les 4 ttys. Ces IRQ sont destinées au processeur, mais elles passent par l'ICU. L'ICU permet de masquer individuellement chaque IRQ et si plusieurs sont levées simultanément alors l'ICU permet de dire quelle est celle prioritaire. La manière dont sont routées les IRQ n'est pas modifiable par logiciel, les IRQ sont des signaux électriques câblés par les architectes. Sur `almol` :

- L'IRQ du timer entre sur l'entrée n°0 de l'ICU.
- Les IRQ de TTY entrent respectivement sur les entrées 10, 11, 12 et 13 de l'ICU.

**Question** : faire un dessin représentant la plateforme avec les signaux IRQ.

### 2. Game over simple

Vous allez réaliser un petit jeu dans lequel vous deviez deviner un nombre tiré au hasard. Nous vous proposons de limiter le temps pendant lequel vous pouvez jouer. Nous allons vous guider pas-à-pas.

Récupérez l'[archive du code du tp2](#), placez-la dans le répertoire `k06-a2` et décompressez-la. Les commandes ci-dessous supposent que vous avez mis l'archive dans le répertoire `k06-a2`

```
cd ~/k06-a2
tar xvfz tp2.tgz
cd tp2/01_gameover
```

Le code de l'application est le suivant (dans `uapp/main.c`)

```
#include <libc.h>
int main (void)
{
    int guess;
    int random;
    char buf[8];
    char name[16];

    fprintf(0, "Tapez votre nom : ");
    fgets(name, sizeof(name), 0);
    if (name[strlen(name)] == '\n')
        name[strlen(name)] = 0;
    srand(clock()); // start the random generator with a "random" seed.

    random = 1 + rand() % 99;
    fprintf(0, "Donnez un nombre entre 1 et 99: ");
    do {
        fgets(buf, sizeof(buf), 0);
        guess = atoi (buf);
        if (guess < random)
            fprintf(0, "%d est trop petit: ", guess);
        else if (guess > random)
            fprintf(0, "%d est trop grand: ", guess);
    } while (random != guess);

    fprintf(0, "\nGagné %s\n", name);
    return 0;
}
```

- Essayez le jeu (dans le répertoire `tp2/01_gameover`) : tapez `make exec` comme vous pouvez le constater, vous avez le temps de jouer.
- Dans la version précédente du gestionnaire de syscall, nous avons masqué les IRQ en écrivant 0 dans le registre `c0_status` (registre \$12 du coprocesseur 0). Cela avait pour conséquence de mettre tout à 0, entre autre le bit `IE`. Il faut modifier ça, parce que sinon, lorsque l'utilisateur demandera à lire le clavier avec l'appel système `fgets()`, l'IRQ venant du timer ne sera jamais prise en compte (`TOD01`), ensuite au retour de la fonction qui réalise l'appel système, il faut masquer les IRQ pour ne pas avoir d'interruption pendant la restauration des registres jusqu'au `eret` qui fait sortir du kernel.

```
addiu $29, $29, -8*4 // context for $31 + EPC + SR + syscall_code + 4 args
mfc0 $27, $14 // $27 <- EPC (addr of syscall instruction)
mfc0 $26, $12 // $26 <- SR (status register)
addiu $27, $27, 4 // $27 <- EPC+4 (return address)
sw $31, 7*4($29) // save $31 because it will be erased
```

```

sw      $27,    6*4($29)      // save EPC+4 (return address of syscall)
sw      $26,    5*4($29)      // save SR (status register)
sw      $2,      4*4($29)      // save syscall code (useful for debug message)
// TODO1: remplacez "mtc0 $0, $12" par 2 autres pour mettre 1 dans les bits c0_sr.HWIO et c0_sr.IE
// vous pouvez utiliser $26
mtc0    $0,     $12           // SR <- kernel-mode without INT (UM=0 ERL=0 EXL=0 IE=0)

la      $26,    syscall_vector // $26 <- table of syscall functions
andi   $2,     $2,    SYSCALL_NR-1 // apply syscall mask
sll     $2,     $2,     2      // compute syscall index (multiply by 4)
addu   $2,     $26,    $2      // $2 <- & syscall_vector[$2]
lw      $2,     ($2)           // at the end: $2 <- syscall_vector[$2]
jalr    $2                          // call syscall function

// TODO2: Il faut mettre 0 dans SR pour masquer les interruptions
lw      $26,    5*4($29)      // get old SR
lw      $27,    6*4($29)      // get return address of syscall
lw      $31,    7*4($29)      // restore $31 (return address of syscall function)
mtc0    $26,    $12           // restore SR
mtc0    $27,    $14           // restore EPC
addiu   $29,    $29,    8*4    // restore stack pointer
eret                      // return : jr EPC with EXL <- 0

```

```

// TODO1: remplacez "mtc0 $0, $12" par 2 autres pour mettre 1 dans les bits c0_sr.HWIO et c0_sr.IE
li      $26,    0x401         // next value of SR
mtc0    $26,    $12           // SR <- kernel-mode with INT (HWIO=1 UM=0 ERL=0 EXL=0 IE=1)

// TODO2: Il faut mettre 0 dans SR pour masquer les interruptions
mtc0    $0,     $12           // SR <- kernel-mode without INT (UM=0 ERL=0 EXL=0 IE=0)

```

3. Ouvrez le fichier `kernel/kinit.c`. Dans cette fonction, on appelle `archi_init()` avec en paramètre un nombre qui va servir de période d'horloge. Le simulateur de la plateforme sur les machines de la PPTI va environ à 3.5MHz. Combien de secondes demande-t-on dans ce code ?

`arch_init(30*3500000);` about 30 secondes with this simulator (3.5MHz)

4. Ouvrez le fichier `kernel/harch.c` et vous allez devoir remplir 3 fonctions pour configurer le timer: `arch_init()`, `timer_init()` et `timer_isr()` (pour trouver ces fonctions cherchez le mot `TODO`)

```

void arch_init (int tick)
{
    // TODO A remplir avec 4 lignes :
    // 1) appel de la fonction timer_init pour le timer 0 avec tick comme période
    // 2) mise à 1 du bit 0 du registre ICU_MASK en utilisant la fonction icu_set_mask()
    // 3) initialisation de la table irq_vector_isr[] vecteur d'interruption avec timer_isr()
    // 4) initialisation de la table irq_vector_dev[] vecteur d'interruption avec 0
}

static void timer_init (int timer, int tick)
{
    // TODO A remplir avec 2 lignes :
    // 1) initialiser le registre period du timer n°timer avec la période tick (reçus en argument)
    // 2) initialiser le registre mode du timer n°timer avec 3 (démarré le timer avec IRQ demandée) si la période est non nulle
}

static void timer_isr (int timer)
{
    // TODO A remplir avec 3 lignes :
    // 1) Acquiescer l'interruption du timer en écrivant n'importe quoi dans le registre resetirq
    // 2) afficher un message "Game Over" avec kprintf()
    // 3) appeler la fonction kernel exit() (c'est une sortie définitive ici)
}

```

```

void arch_init (int tick)
{
    timer_init (0, tick);           // sets period of timer n'0 (thus for CPU n'0) and starts it
    icu_set_mask (0, 0);           // [CPU n'0].IRQ <-- ICU.PIN[0] <- Interrupt signal timer n'0
    irq_vector_isr [0] = timer_isr; // tell the kernel which isr to exec for ICU.PIN n'0
    irq_vector_dev [0] = 0;         // device instance attached to ICU.PIN n'0
}

static void timer_init (int timer, int tick)
{
    __timer_regs_map[timer].period = tick; // next period
    __timer_regs_map[timer].mode = (tick)?3:0; // timer ON with IRQ only if (tick != 0)
}

static void timer_isr (int timer)
{
    __timer_regs_map[timer].resetirq = 1; // IRQ acknowledgement to lower the interrupt signal
    kprintf ("Game Over\n");
    exit(1);
}

```

### 3. Game over avec décompteur

Dans ce qui précède, l'exécution de l'ISR du Timer est fatale puisqu'elle stoppe l'application après l'affichage de "Game Over!". Nous vous proposons de modifier l'ISR afin d'avoir un comportement un peu plus réaliste. Dans cette nouvelle version, l'ISR du timer décrémente un compteur alloué dans une variable globale du noyau puis elle revient dans l'application tant que ce compteur est différent de 0. Donc, dans l'ISR du timer si le compteur est différent de 0, elle affiche un message avec la valeur du compteur, sinon elle affiche "game over!" et stoppe l'application, comme dans l'exercice précédent.

Par exemple, au lieu d'afficher:

```

|_|_/_v'\_/_/_
|/_/(\_/_)/\_/_
|_\_x\_/_/_/_

```

Tapez votre nom : Moi  
 Donnez un nombre entre 1 et 99 : 45  
 45 est trop grand: 20  
 20 est trop grand:  
 0 est trop petit:

```
Game Over
[105002991] EXIT status = 1
```

l'application pourrait afficher:

$$\begin{array}{c} | \quad | \quad / \text{'v'} \backslash \quad / \quad / \\ | \quad / \quad / ( \quad ) \quad / \quad - \quad \backslash \\ | \quad \backslash \quad \backslash \quad x \quad x \quad \backslash \quad - \quad / \end{array}$$

```
Tapez votre nom : Moi
Donnez un nombre entre 1 et 99: 45
45 est trop grand: 20
20 est trop grand:
..3 : 12
12 est trop petit: 15
15 est trop petit:
..2 :
..1 :
Game Over
[115002778] EXIT status = 1
```

```
kernel/harch.c
```

```
extern void arch_init (int tick, unsigned quantum);
```

```
kernel/harch.c
```

```
static unsigned timer_quantum;
static void timer_init (int timer, int tick, unsigned quantum)
{
    __timer_regs_map[timer].resetirq = 0;           // to delete previous untratted IRQ
    __timer_regs_map[timer].period = tick;          // next period
    __timer_regs_map[timer].mode = (tick)?3:0;      // timer ON with IRQ only if (tick != 0)
    timer_quantum = quantum % 100;                  // %100 to avoid aberrant value
}
static void timer_isr (int timer)
{
    __timer_regs_map[timer].resetirq = 1;
    if (timer_quantum) {
        kprintf ("\n...%d : ", timer_quantum--);
    } else {
        kprintf ("\nGame Over\n");
        exit(1);
    }
}
```

```
kernel/kinit.c
```

```
void kinit (void)
{
    [...]
    arch_init (3*3500000, 10); // tick is about 3 seconde, quantum is then about 30 secondes
    [...]
}
```

#### 4. Évaluation de la durée d'une ISR

Dans cet usage du TIMER, les ISR ne sont pas fatales, sauf la dernière. En utilisant le mode debug (make debug) et le fichier `trace0.S`, déterminez la durée en cycles du traitement par le noyau d'une IRQ du timer. Ce n'est pas exactement la même durée pour toutes les IRQ.

Pour cette question, il faut commenter les affichages dans l'ISR, changer la valeur du tick pour voir plus d'IRQ (par exemple 1000) et exécuter en mode debug puis regarder la trace dans `trace0.S`, il faut chercher un `kentry` correspondant à une IRQ et chercher l'instruction `eret` qui marque la fin du traitement.

- La durée mesurée est de l'ordre de 430 cycles pour la première (vers le cycle 3600).

*Dernière modification le 3 févr. 2023 à 15:59:15)*