1. Architecture externe du Processeur MIPS32
    1. A) INTRODUCTION
    2. B) REGISTRES VISIBLES DU LOGICIEL
        1. 1) Registres non protégés
        2. 2) Registres protégés
    3. C) ADRESSAGE DE LA MÉMOIRE
        1. 1) Espace d'adressage : adressage par octet
        2. 2) Calcul d'adresse
        3. 3) Protection mémoire
    4. D) JEU D'INSTRUCTIONS
        1. 1) Généralités et format des instructions
        2. 2) Codage des instructions
        3. 3) Jeu d'instructions
    5. E) EXCEPTIONS / INTERRUPTIONS / APPELS SYSTÈME
        1. 1) Exceptions
        2. 2) Interruptions
        3. 3) Appels système: instructions `syscall` et `break`
        4. 4) Signal `RESET`
        5. 5) Sortie du noyau
        6. 6) Gestion du registre d'état `c0_sr`
        7. 7) Gestion du registre de cause `c0_cause`

Architecture externe du Processeur MIPS32

? Version 3.0
? Septembre 2020
? Alain Greiner (auteur initial)
? Module AS5 Archi L3 S5

A) INTRODUCTION

Ce document présente une version simplifiée de l'architecture externe du processeur MIPS32. Pour des raisons de simplicité, tous les mécanismes matériels de gestion de la mémoire virtuelle ont été délibérément supprimés.

L'architecture externe représente ce que doit connaître un utilisateur souhaitant programmer en assembleur, ou souhaitant écrire un compilateur. Elle définit:

- Les registres visibles du logiciel ;
- L'adressage de la mémoire ;
- Le jeu d'instructions ;
- Les mécanismes de traitement des interruptions, des exceptions et appels système.

Le processeur MIPS32 est un processeur 32 bits conçu dans les années 1980. Son jeu d'instructions est de type

RISC (Reduced Instruction Set Computer). Il existe de nombreuses réalisations industrielles de cette architecture (SIEMENS, NEC, LSI LOGIC, SILICON GRAPHICS, MICROCHIP, etc.).

Cette architecture est suffisamment simple pour présenter les principes de base de l'architecture des processeurs, et suffisamment puissante pour supporter un système d'exploitation multitâches tel qu'UNIX, puisqu?il supporte deux modes de fonctionnement utilisateur (*user*) et système (*kernel*).

- Dans le mode *user*, certaines régions de la mémoire et certains registres du processeur sont protégés et donc inaccessibles.
- Dans le mode *kernel*, toutes les ressources sont accessibles, c'est-à-dire toute la mémoire et tous les registres.

L'architecture interne n'est pas présentée dans ce module. Elle dépend des choix de réalisation matérielle. Plusieurs implémentations matérielles ont été réalisées à Sorbonne Université dans un but d'enseignement et de recherche : une version microprogrammée, simple mais peu performante (1 instruction s'exécute en 4 cycles d'horloge), une version pipeline plus performante (1 instruction par cycle) mais plus complexe et une version superscalaire, encore plus performante (2 instructions par cycle) mais beaucoup plus complexe.

La spécification du langage d'assemblage, les conventions d?utilisation des registres, ainsi que les conventions d?utilisation de la pile font l'objet d'un document séparé.

B) REGISTRES VISIBLES DU LOGICIEL

Tous les registres visibles du logiciel, c'est-à-dire ceux dont la valeur peut être lue ou modifiée par les instructions, sont des registres 32 bits. Il y a deux catégories de registres, protégés et non protégés, dont l'accès dépend du mode d'exécution du processeur. En mode *user*, seuls les registres non protégés sont accessibles. En mode *kernel*, tous les registres sont accessibles.

1) Registres non protégés

Le processeur possède 35 registres utilisables par les instructions standards (c'est-à-dire les instructions qui peuvent s'exécuter aussi bien en mode *user* qu'en mode *kernel*).

Les registres GPR

Il y a 32 registres généraux (GPR signifie *General Purpose Register*) numétotés de `$0` à `$31`
Ces registres sont directement utilisés par les instructions et permettent de stocker des résultats de calculs intermédiaires.

Le registre `$0` est particulier:
? la lecture fournit la valeur constante `0x00000000`
? l?écriture ne modifie pas son contenu.

Le registre `$31` est utilisé par les instructions d'appel de fonctions pour sauvegarder l'adresse de retour.
? Les instructions d'appel de fonctions sont : `bgezal`, `bltzal`, `jal`et `jalr`

Le registre `PC`

C'est le *Program Counter* ou compteur ordinal en français.
Ce registre contient l'adresse de l'instruction en cours d'exécution. Sa valeur est modifiée par toutes les instructions.

Les registres `HI` et `LO`

Ces sont les registres de la multiplication entière et de la division Euclidienne.
Ces deux registres 32 bits sont utilisés pour stocker le résultat d'une multiplication ou d'une division.
La multiplication de deux nombre de 32 bits est un mot de 64 bits (poids forts dans `HI` et poids faibles dans `LO`).
La division Euclidienne de deux nombres 32 bits produit un quotient sur 32 bits dans `LO` et un reste sur 32 bits dans `HI`.

2) Registres protégés

L'architecture du MIPS32 définit 32 registres protégés (numérotés également de `$0` à `$31` mais ils sont dans un coprocesseur). Ces registres protégés ne sont accessibles, en lecture comme en écriture, que par les instructions privilégiées `mtc0` et `mfc0` ne pouvant être exécutées qu'en mode *kernel*. `mtc0` et `mfc0` signifient respectivement *Move-To-Coprocessor-0* et *Move-From-Coprocessor-0*. En effet, ces registres appartiennent au "coprocesseur système" n°0 (appélé aussi `c0` pour *Coprocessor 0*). En pratique, cette version du processeur MIPS32 en définit 6. Ils sont utilisés par le système d?exploitation pour la gestion des interruptions, des exceptions et des appels système (voir chapitre E).

Le registre `c0_sr`

Le registre `sr` de `c0` est le registre d'état (*Status Register*). Il contient en particulier le bit qui définit le mode : *user* ou *kernel*, ainsi que les bits de masquage des interruptions.
Ce registre a le numéro `$12`.

Le registre `c0_cause`

Le registre `cause` de `c0` est le registre de cause (*Cause Register*). En cas d'interruption, d'exception ou d'appel système, le code en cours d'exécution par le processeur est dérouté vers le noyau du système d'exploitation. Le contenu de `c0_cause` définit la cause d'appel du noyau.
Ce registre a le numéro `$13`.

Le registre `c0_epc`

Le registre `epc` de `c0 est le registre d'exception (*Exception Program Counter*). Il contient soit l'adresse de retour (PC + 4) en cas d'interruption, soit l'adresse de l'instruction courante (`pc`) en cas d'exception ou d'appel système.
Ce registre a le numéro `$14`.

Le registre `c0_bar`

Le registre `bar` de `c0` est legistre d'adresse illégale (*Bad Address Register*). En cas d'exception de type "adresse illégale", il contient la valeur de l'adresse mal formée.
Ce registre a le numéro `$8`.

Le registre `c0_procid`

Le registre `procid` est le registre en lecture seulement contenant le numéro du processeur. Cet index « cablé » est utilisé par le noyau du système d?exploitation pour gérer des architectures multiprocesseurs.

Ce registre possède le numéro `$15`.

Le registre `c0_count`

Le registre `count` de `c0` est le registre en lecture seulement contenant le nombre de cycles exécutés depuis l?initialisation du processeur.

Ce registre possède le numéro `$16`.

Comportement des instructions `mtc0`et `mfc0`

instruction assembleur	comportement dans le processeur	Remarques
mtc0 $gpr, $c0	Copro. 0 ($C0) <--- Reg. GPR ($gpr)	$c0 = $8, $12, $13, $14, $15 ou $16 ; $gpr = $0 .. $31
mfc0 $gpr, $c0	Reg. GPR ($gpr) <--- Copro. 0 ($C0)	$c0 = $8, $12, $13, $14, $15 ou $16 ; $gpr = $0 .. $31

C) ADRESSAGE DE LA MÉMOIRE

1) Espace d'adressage : adressage par octet

L'ensemble des adresses que peut former le processeur définit son espace d'adressage. Toutes les adresses formées sont des adresses d'octets, ce qui signifie que la mémoire est vue par le processeur comme un tableau d'octets qui contient aussi bien les données que les instructions.

Les adresses sont codées sur 32 bits. Les instructions sont codées sur 32 bits. Les échanges de données avec la mémoire se font par mot de 32 bis (4 octets consécutifs), ou par demi-mot 16 bits (2 octets consécutifs) ou par octet (8 bits). Pour les transferts de mots et de demi-mots, le processeur respecte la convention "little endian" (l'octet de poids faible est à l'adresse à plus petite).

L'adresse d'un mot (4 octets) de donnée ou d'instruction doit être multiple de 4. L'adresse d'un demi-mot doit être multiple de 2, on dit que les adresses doivent être "alignées". Le processeur part en exception si une instruction calcule une adresse qui ne respecte pas cette contrainte. Plus généralement, une donnée est alignée en mémoire, si l'adresse de son premier octet (à l'adresse la plus petite) est un multiple de sa taille.

2) Calcul d'adresse

Le mode d'adressage définit la manière dont le processeur calcule les adresses de la mémoire pour y accéder en lecture ou en écriture. Il n'existe qu'un seul et unique mode d?adressage pour le MIPS32. Il consiste à effectuer la somme entre le contenu d'un registre général (GPR) $i, défini dans l'instruction, et d'un déplacement qui est une valeur immédiate signée, sur 16 bits (de -32768 à +32767), contenue également dans l'instruction:

```
adresse = Ri + Déplacement
```

3) Protection mémoire

L?espace d'adressage de la mémoire est découpé en 2 parties identifiés par le bit de poids fort de l?adresse :

```
bit n°31 de l'adresse = 0        ==>     partie utilisateur
bit n°0 de l'adresse  = 1        ==>     partie système
```

Quand le processeur est en mode système alors les 2 parties (utilisateur et système) sont accessibles. Quand le processeur est en mode utilisateur alors seule la partie utilisateur est accessible.

Quand le processeur est en mode utilisateur, si une instruction essaie d'accéder à la mémoire avec une adresse de la partie système alors le processeur part en exception, c'est-à-dire que le programme est dérouté vers le noyau du système d'exploitation (voir section E)

Si une anomalie est détectée au cours du transfert entre le processeur et la mémoire, alors le système mémoire le signale ce qui déclenche également un départ en exception.

D) JEU D'INSTRUCTIONS

1) Généralités et format des instructions

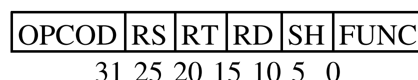
Le processeur possède 57 instructions qui se répartissent en 4 classes :

- 33 instructions arithmétiques/logiques entre registres
- 12 instructions de branchement
- 7 instructions de lecture/écriture mémoire
- 5 instructions système

Toutes les instructions ont une longueur de 32 bits et possèdent l'un des trois formats suivants R, I ou J:

Format R

Le format R est utilisé par les instructions ayant 2 registres sources (désignés par RS et RT) et un registre résultat désigné par RD.
La forme générale est `OPCOD RD, RS, RT` dont le comportement est `RD <- RS OPCOD RT`.
Par exemple `sub $4,$8,$16` réalise `$4 <- $8 - $16`.
Codage:



OPCOD	RS	RT	RD	SH	FUNC

31 25 20 15 10 5 0

Format I

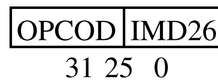
Le format I est utilisé (i) par les instructions de lecture/écriture mémoire, (ii) par les instructions utilisant un opérande immédiat, (iii) ainsi que par les branchements courte distance (conditionnels).
La forme générale est `OPCOD RT, RS, IMD16` dont le comportement est `RT <- RS OPCOD IMM16`.
Par exemple `addi $4,$8,-42` réalise `$4 <- $8 - 42` ou `lb $4,42($8)` réalise `$4 <- MEM[$8 + 42]`

Codage:

OPCOD	RS	RT	IMD16
31	25	20	15 0

Format J

Le format J n?est utilisé que pour les branchements inconditionnels longue distance La forme générale est `OPCOD IMD26` dont le comportement est `PC <- PC+IMD26`.
Par exemple `j 0x40` réalise `PC <- PC + 0x40` (notez que l'argument de l'instruction est presque toujours une étiquette du programme et que c'est le programme d'assemblage qui calcule la valeur IMD26).
Codage:

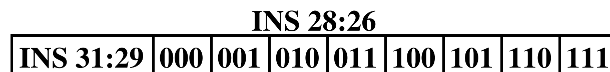


OPCOD	IMD26
31	25 0

2) Codage des instructions

Le codage des instructions est principalement défini par les 6 bits du champ code opération (appelé OPCOD) de l'instruction (INS 31:26). Cependant, trois valeurs particulières de ce champ définissent en fait une famille d'instructions : il faut alors analyser d'autres bits de l'instruction pour décoder l'instruction. Ces codes particuliers sont : SPECIAL (valeur "000000"), BCOND (valeur "000001") et COPRO (valeur "010000")

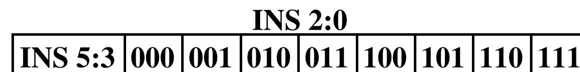
- codage du champ OPCOD:



	INS 28:26							
INS 31:29	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**

Par exemple, ce tableau indique que l'instruction LHU a un OPCOD à "100101".

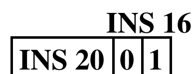
- Lorsque l'OPCODE a la valeur SPECIAL ("000000"), il faut analyser les 6 bits de poids faible de l'instruction (INS 5:0):



	INS 2:0							
INS 5:3	**000**	**001**	**010**	**011**	**100**	**101**	**110**	**111**

Par exemple, l'instruction MTLO a un OPCOD à "000000" et le champ FUNC est à "010011".

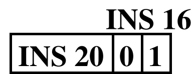
- Lorsque l'OPCOD a la valeur BCOND, il faut analyser les bits 20 et 16 de l'instruction.



	INS 16	
INS 20	**0**	**1**

Par exemple, l'instruction BTLZAL a un OPCOD à "000001",
le bit IN20 est à "1" et le bit INS16 est à "0".

* Lorsque l'OPCOD a la valeur COPRO, il faut analyser les bits 25 et 23 de l'instruction. Les trois instructions de cette famille COPRO sont des instructions privilégiées. Remarquez que ERET à deux codages.



Par exemple, l'instruction MTC0 a un OPCOD à "010000",
le bit IN20 est à "0" et le bit INS16 est à "0".

3) Jeu d'instructions

Le jeu d'instructions est "orienté registres". Cela signifie que les instructions arithmétiques et logiques prennent leurs opérandes dans des registres et rangent leur résultat dans un registre. Les seules instructions permettant de lire ou d'écrire des données en mémoire effectuent un simple transfert entre un registre général et la mémoire, sans aucun traitement arithmétique ou logique.

La plupart des instructions arithmétiques et logiques se présentent sous les 2 formes registre-registre et registre-immédiat:

instruction assembleur	comportement dans le processeur	format d'instruction
add rd, rs, rt	R(rd) <--- R(rs) op R(rt)	*format R*
addi rd, rs, imm	R(rd) <--- R(rs) op IMD16	*format I*

L?opérande immédiat 16 bits est signé pour les opérations arithmétiques et non signé pour les opérations logiques.

Le déplacement est de 16 bits pour les instructions de branchement conditionnelles (Bxxx) et de 26 bits pour les instructions de saut inconditionnelles (Jxxx). De plus les instructions JAL, JALR, BGEZAL, et BLTZAL sauvegardent une adresse de retour dans le registre `$31`. Ces instructions sont utilisées pour les appels de sous-programme.

Toutes les instructions de branchement conditionnel sont relatives au PC (compteur ordinal) pour que le code soit translatable (déplaçable ailleur en mémoire), c'est-à-dire que l'adresse de branchement est le résultat de l'addition entre la valeur du compteur ordinal et un déplacement signé.

Les instructions `mtc0` (*move to c0*) et `mfc0` (*move from c0*) permettent de transférer le contenu des registres `c0_sr`, `c0_cause`, `c0_epc` etc. vers un registre général GPR et inversement. Ces 2 instructions ne peuvent être exécutées qu?en mode système, de même que l'instruction `eret` qui permet de restaurer l'état antérieur du registre d'état `c0_sr` avant de sortir du gestionnaire d'exceptions.

E) EXCEPTIONS / INTERRUPTIONS / APPELS SYSTÈME

Il existe quatre types d'évènements qui peuvent interrompre l'exécution "normale" d'un programme:

* les exceptions ;
* les interruptions ;

- les appels système (instructions `syscall` et `break`)
- le signal RESET.

Dans tous ces cas, le principe général consiste à dérouter le programme vers un code spécial (appelée noyau du système d'exploitation) qui s'exécute en mode système et à qui il faut transmettre les informations minimales lui permettant de traiter le problème.

1) Exceptions

Les exceptions sont des évènements "anormaux" détectés au moment de l'exécution des instructions. Ils sont le plus souvent liés à une erreur de programmation qui empêche l'exécution correcte de l'instruction en cours. La détection d'une exception entraîne l'arrêt immédiat de l'exécution de l'instruction fautive. Ainsi, on assure que l'instruction fautive ne modifie pas la valeur d'un registre visible ou de la mémoire. Les exceptions ne sont évidemment pas masquables, cela signifie que l'on ne peut pas interdire leur gestion. Il y a 7 types d'exception dans cette version du processeur MIPS32 :

ADEL
: Adresse illégale en lecture : adresse non alignée ou se trouvant dans la partie système alors que le processeur est en mode utilisateur.

ADES
: Adresse illégale en écriture : adresse non alignée ou accès à une donnée dans le partie système alors que le processeur est en mode utilisateur.

DBE
: Data bus erreur : le système mémoire signale une erreur en activant le signal BERR à la suite d'un accès de donnée à une adresse qui n'a pas de case mémoire associée. On dit qu'elle n'est pas *mappée*. Cette erreur est aussi nommée erreur de segmentation ('segmentation fault` en anglais).

IBE
: Instruction bus erreur : le système mémoire signale une erreur en activant le signal BERR à l'occasion d'une lecture instruction. C'est le même problème que pour **DBE** mais cela concerne les instructions.

OVF
: Dépassement de capacité : lors de l'exécution d'une instruction arithmétique (ADD ou ADDI), le résultat ne peut être représenté sur 32 bits. Par exemple, la somme de 2 nombres positifs donne un nombre négatif.

RI
: OPCOD illégal : l'OPCOD ne correspond à aucune instruction connue, il s'agit probablement d'un branchement dans une zone mémoire ne contenant pas du code exécutable.

CPU
: Coprocesseur inaccessible : tentative d'exécution d'une instruction privilégiée (`mtc0`, `mfc0`, `eret`) alors que le processeur est en mode utilisateur.

Dans tous les cas, le processeur doit passer en mode système et se brancher au noyau du système d'exploitation implanté à l'adresse `0x80000180`. Après avoir identifié que la cause est une exception (en examinant le contenu du registre `c0_cause`), le noyau se branche alors au gestionnaire d?exception. Ici, toutes les exceptions sont fatales, il n'y a pas de reprise de l'exécution de l'application contenant l'instruction fautive. Le processeur doit cependant transmettre au gestionnaire d'exceptions l'adresse de l'instruction fautive et indiquer dans le registre de cause le type d'exception détectée. Lorsqu?il détecte une exception, le matériel doit donc:

- sauvegarder `PC` (l'adresse de l'instruction fautive) dans le registre `c0_epc` ;
- passer en mode système et masquer les interruptions dans `c0_sr` ;
- sauvegarder éventuellement l?adresse fautive dans `c0_bar`;
- écrire le type de l'exception dans le registre `c0_cause`;

- brancher à l'adresse `0x80000180`.

2) Interruptions

Les requêtes d'interruption matérielles sont des évènements asynchrones provenant des contrôleurs de périphériques. Elles peuvent être masquées (ignorées) par le processeur. Le processeur MIPS32 possède 6 lignes d'interruptions externes qui peuvent être masquées globalement ou individuellement. L'activation d'une de ces lignes est une requête d'interruption. Elles sont notifiées dans le registre `c0_cause` et, si elles ne sont pas masquées, elles sont prises en compte à la fin de l'exécution de l'instruction en cours. Cette requête doit être maintenue active par le contrôleur de périphérique tant qu'elle n'a pas été prise en compte par le processeur.

Le processeur doit alors passer alors en mode système et se brancher au noyau du système d'exploitation. Après avoir identifié que la cause est une interruption (en examinant le contenu du registre `c0_cause`), le noyau se branche au gestionnaire d?interruption qui doit appeler une fonction appropriée pour le traitement de la requête. Cette fonction est appelée routine d?interruption ou ISR (pour *Interrupt Service Routine*). Comme il faut reprendre l'exécution de l'application en cours à la fin du traitement de l'interruption, il faut sauvegarder une adresse de retour. Lorsqu?il reçoit une requête d?interruption non masquée, le matériel doit donc :

- sauvegarder PC+4 (l'adresse de retour) dans le registre `c0_epc` ;
- passer en mode système et masquer les interruptions dans `c0_sr` ;
- écrire qu'il s'agit d'une interruption dans le registre `c0_cause` ;
- brancher à l'adresse `0x80000180`program.

En plus des 6 lignes d'interruption matérielles, le processeur MIPS32 possède un mécanisme d'interruption logicielle: Il existe 2 bits dans le registre de cause `c0_cause` qui peuvent être écrits par le logiciel au moyen de l'instruction privilégiée `mtc0`. La mise à 1 de ces bits déclenche le même traitement que les requêtes d'interruptions externes, s'ils ne sont pas masqués.

3) Appels système: instructions `syscall` et `break`

L'instruction `syscall` permet à une application de l'utilisateur de demander un service au noyau du système d'exploitation, comme par exemple effectuer une entrée-sortie. Le code définissant le type de service demandé au système, et d'éventuels paramètres doivent avoir été préalablement rangés dans des registres généraux. L'instruction `break` est utilisée plus spécifiquement pour poser un point d'arrêt (dans un but de debuggage du logiciel): on remplace brutalement une instruction du programme à debugger par l'instruction `break`. Dans les deux cas, le processeur passe en mode système et se branche ici encore au noyau. Après avoir identifié que la cause est un appel système (en examinant le contenu du registre `c0_cause`), le noyau se branche au gestionnaire d?appels système. Lorsqu?il rencontre une des deux instructions `syscall` ou `break`, le matériel effectue les opérations suivantes :

- sauvegarder PC (l'adresse de l'instruction) dans le registre `c0_epc` (l?adresse de retour est PC + 4) ;
- passer en mode système et masquage des interruptions dans `c0_sr` ;
- écrire la cause du déroutement dans le registre `c0_cause` ;
- brancher à l'adresse `0x80000180`.

4) Signal `RESET`

Le processeur possède également une entrée `RESET` dont l'activation pendant au moins un cycle entraîne le branchement inconditionnel du code de démarrage de l'ordinateur (code de boot). Ce code, implanté à l?adresse `0xBFC00000` doit principalement charger le code du noyau du système d?exploitation dans la mémoire depuis le

disque ou le réseau et se brancher à la fonction d'initialisation du noyau. Cette fonction initialise les contrôleurs de périphériques et les structures internes du noyau et, à la fin elle se branche à la première application utilisateur. Dans notre modèle d'ordinateur, le noyau est pré-chargé en mémoire et le code de boot se contente d'appeler la fonction d'initialisation.

Cette requête est très semblable à une septième ligne d'interruption externe avec les différences importantes suivantes:

- elle n'est pas masquable :
- il n'est pas nécessaire de sauvegarder une adresse de retour.
- le gestionnaire de reset est implanté à l'adresse "0xBFC00000".

Dans ce cas, le processeur doit :

- passer en mode système et masque les interruptions dans SR
- brancher à l'adresse "0xBFC00000"

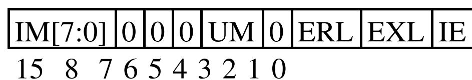
5) Sortie du noyau

Avant de reprendre l'exécution de l'application qui a effectué un appel système (instructions `syscall`) ou qui a été interrompu par une interruption, il est nécessaire d'exécuter l'instruction `eret`. Cette instruction modifie le contenu du registre `c0_sr`, et effectue un branchement à l?adresse contenue dans le registre `c0_epc`.

6) Gestion du registre d'état `c0_sr`

Le registre `c0_sr` contient l'état du processeur. Cela concerne son comportement vis-à-vis des requêtes d'interruptions, c'est-à-dire les masques des interruptions matérielles et logicielles, et le mode d'exécution, mode système (*kernel*) ou en mode utilisateur (*user*).

- La figure suivante présente le contenu des 16 bits de poids faible du registre `c0_sr`. Cette version du MIPS32 n?utilise que 12 bits:



`IE`	Interrupt Enable	0 = toutes les interruptions sont masquées 1 = interruptions non-masquées mais elles peuvent l'être avec `IM[7:0]`
`EXL`	Exception Level	0 = aucun effet sur le processeur 1 = le processeur vient d'entrer dans le noyau et donc le processeur est en mode *kernel* et interruptions massquées
`ERL`	Error Level	1 = après le signal reset ou certaines erreurs de la mémoire
`UM`	User Mode	0 = mode d'exécution *kernel* 1 = mode d'exécution *user*
`IM![7:0]`	Masques individuels	pour les six lignes d?interruption matérielles (bits `IM[7:2]`) et pour les 2 interruptions logicielles (bits `IM[1:0]`)

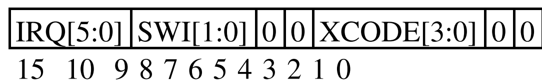
- Quelques remarques :
  - ♦ Le processeur a le droit d?accéder aux ressources protégées (registres du coprocessor 0 `c0`), et aux adresses mémoires >= `0x80000000`) si et seulement si le bit `UM` vaut `0`, ou si l?un des deux bits `ERL` et `EXL` vaut `1`.
  - ♦ Les interruptions sont autorisées si et seulement si le bit `IE` vaut `1`, et si les deux bits `ERL` et `EXL` valent `00`, et si le bit correspondant de `IM` vaut `1`.
  - ♦ Les trois types d?événements qui déclenchent le branchement au noyau: (interruptions, exceptions et appels système) forcent le bit `EXL` à `1`, ce qui masque les interruptions et autorise l?accès aux ressources protégées.
  - ♦ L?activation du signal `RESET` qui force le branchement au code de boot force le bit `ERL` à `1`, ce qui masque les interruptions et autorise l?accès aux ressources protégées.
  - ♦ L?instruction `eret` force le bit `EXL` à `0`.

- Lors de l?activation du RESET :
  - ♦ `c0_sr` contient donc la valeur `0x0004` (`0b0000000000000100`).
  - ♦ Pour exécuter un programme utilisateur en mode protégé, avec interruptions activées, il doit contenir la valeur `0xFF11`.
  - ♦ Le noyau doit écrire la valeur `0xFF13` dans `c0_sr` et l?adresse de la première fonction du programme utilisateur dans `c0_epc` avant d?appeler l?instruction `eret`.

7) Gestion du registre de cause `c0_cause`

Le registre `c0_cause` contient trois champs. Les 4 bits du champ `XCODE[3:0]` définissent la cause de l'appel du noyau. Les 6 bits du champ `IRQ[5:0]` représentent l'état des lignes d'interruption externes au moment de l'appel su noyau. Les 2 bits `SWI[1:0]` représentent les requêtes d'interruption logicielle.

- La figure suivante montre le format du registre de cause CR :



- Les valeurs possibles du champ `XCODE` sont les suivantes :

0000	**INT**	Interruption
0001		Inutilisé
0010		Inutilisé
0011		Inutilisé
0100	**ADEL**	Adresse illégale en lecture
0101	**ADES**	Adresse illégale en écriture
0110	**IBE**	Bus erreur sur accès instruction
0111	**DBE**	Bus erreur sur accès donnée
1000	**SYS**	Appel système (`syscall`)
1001	**BP**	Point d'arrêt (`break`)
1010	**RI**	OPCOD illégal
1011	**CPU**	Coprocesseur inaccessible

1100	**OVF**	Overflow arithmétique
1101		Inutilisé
1110		Inutilisé
1111		Inutilisé