Documentation MIPS Architecture et assembleur (mode kernel)

Ce document est la suite du document Documentation MIPS32 architecture et assembleur (mode user) (*Ce document est tiré du document initialement écrit par Alain Greiner*).

1. Modes d'exécution du processeur MIPS

Le MIPS 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. C'est dans ce mode que s'exécute les applications.
- Dans le mode *kernel*, toutes les ressources sont accessibles, c'est-à-dire toute la mémoire et tous les registres. Dans ce mode, toutes les instructions sont autorisées, à la fois les instructions standards (`add`, `or`, `lw`, `mul`, etc.), mais aussi les ***instructions protégées*** qui permettent de contrôler l'état de fonctionnement du processeur. C'est dans ce mode que s'exécute le noyau du système d'exploitation.

Ce document détaille les éléments de l'architecture externe du processeur et du langage d'assemblage spécifique au mode *kernel*.

2. Registres protégés utilisables seulement en mode kernel

En mode *kernel*, tous les « registres » sont accessibles, à la fois les registres non protégés et aussi les registres protégés. Pour rappel, les registres non protégés sont les GPR (`$0` à `$31`), le registre `PC` (accessible implicitement avec les instructions de branchement) et les registres `HI` et `LO`. Les registres non protégés sont destinés au calcul alors que les ***registres protégés*** sont destinés au contrôle de l'état du processeur.

L'architecture du MIPS32 définit 32 registres protégés, numérotés de `$0` à `$31`, c'est-à-dire comme les registres GPR mais ils ont des instructions d'accès spécifiques. En effet, ces registres protégés ne sont accessibles que par des instructions protégées présentées dans la section 4.

Ces registres appartiennent au "coprocesseur système" n°0 (appélé aussi `c0` pour *Coprocessor 0*). Dans cette version du processeur MIPS32, il y en a 6. Ils sont tous utilisés par le système d'exploitation pour la gestion des interruptions, des exceptions et des appels système. Dans ce document, nous ferons précéder le numéro du registre protégé par `C0_` afin de ne pas les confondre avec les registres standards.

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 d'exécution du processeur: *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 programme en cours d'exécution est dérouté vers le noyau du système d'exploitation. Le contenu de `C0_CAUSE` contient un code qui 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 : (i) soit l'adresse de retour (PC + 4) en cas d'interruption ; (ii) 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 le registre d'adresse illégale (*Bad Address Register*). En cas d'exception de type *adresse illégale*, il contient la valeur de l'adresse mal formée. Une adresse est illégale, par exemple, si vous tentez une lecture de mot (`lw`) a une adresse non alignée (non multiple de 4) ou si vous tentez une lecture en dehors des segments d'adresse où se trouve de la mémoire.
Ce registre a le numéro `$8`.

Le registre `C0_PROCID`
Le registre `PROCID` de `C0` est un registre en lecture seule contenant le numéro du processeur. Cet index « cablé » est utilisé par le noyau du système d'exploitation. Il n'a de sens que pour gérer des architectures multiprocesseurs (multicore).
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 `$9`.

3. Découpage de l'espace d'adressage

L'espace d'adressage de la mémoire est découpé en 2 parties identifiées par le bit de poids fort de l'adresse (bit n°31). Quand le processeur est en mode *kernel* alors les 2 parties (protégée et non protégée) sont accessibles. Quand le processeur est en mode *user* alors seule la partie protégée est accessible.

```
Bit n°31 de l'adresse = 0       partie non protégée utilisable dans tous les modes du processeur
                                destinée au programme de l'utilisateur
Bit n°31 de l'adresse = 1       partie protégée utilisable seulement en mode kernel
                                réservée au noyau du système d'exploitation
```

Quand le processeur est en mode *user*, si une instruction essaie d'accéder à la mémoire avec une adresse de la partie *protégée* alors le processeur part en exception, c'est-à-dire que le programme fautif est dérouté vers le noyau du système d'exploitation.

4. Instructions protégées

La version du MIPS32 que nous utilisons possède une cinquantaine d'instructions, il y a les instructions standards utilisables quel que soit le mode d'exécution du processeur et il y a les instructions protégées qui ne sont utilisables qu'en mode *kernel*. Les instructions standards sont présentées dans le document sur l'architecture et l'assembleur en mode user. Ce sont les instructions arithmétiques/logiques entre registres, les instructions de branchement, les instructions de lecture et écriture mémoire et l'instruction `syscall`. Nous utilisons 3 instructions protégées, utilisables donc seulement en mode *kernel* : `mtc0`, `mfc0` et `eret`.

`mtc0` et `mfc0`

Elles signifient respectivement *Move-To-Coprocessor-0* et *Move-From-Coprocessor-0*. Comme leur nom l'indique, elles permettent de déplacer le contenu des registres entre les bancs (GPR et Copro).

instruction assembleur	comportement dans le processeur	Remarques
`mtc0 $GPR, $C0`	COPRO. 0 (`$C0`) ← GPR (`$GPR`)	`$C0` = `$8` , `$12` , `$13` , `$14` , `$15` OU `$9` `$GPR` = `$0` ... `$31`
`mfc0 $GPR, $C0`	GPR (`$GPR`) ← COPRO. 0 (`$C0`)	`$C0` = `$8` , `$12` , `$13` , `$14` , `$15` OU `$9` `$GPR` = `$0` ... `$31`

`eret`

Elle signifie *Exception-RETurn*, c'est-à-dire *retour d'une exception*. Nous allons voir en détail ce que cela signifie dans la section **5**. Pour le moment, comprenez que c'est l'unique instruction permettant de sortir du mode *kernel* pour entrer ou retourner dans le mode *user*.

instruction assembleur	comportement dans le processeur	Remarques
`eret`	`PC` ← `CO_EPC` `C0_SR.EXL` ← `0`	copie le registre `C0_EPC` (`C0_$14`) dans le registre `PC` et met `0` dans le bit `EXL` du registre `C0_SR` (bit n°1 de `C0_$12` <— `0`)

Codage des instructions protégées

Elles utilisent toutes le format R avec le champ `OPCOD` à la valeur `COPRO` (c.-à-d. `0b010000`). L'instruction est alors codée avec les bits 25 et 23 de l'instruction (ces deux bits sont dans le champ `RS`). Remarquez que `eret` à deux codages.

OPCOD	RS	RT	RD	SH	FUNC
31	25	20	15	10	5 0

INS 23

INS 25	0	1
0	`mfc0`	`mtc0`
1	`eret`	

Pour les instructions `mtc0` et `mfc0`, le premier argument est mis dans le champ `RT` et le second argument est mis dans le champ `RD` .

instruction	comportement	commentaire
`mtc0 RT, RD`	`C0_RD` ← `RT`	Recopie le contenu du registre GPR n° `RT` dans le registre n° `RD` du coprocesseur 0
`mfc0 RT, RD`	`RT` ← `C0_RD`	Recopie le contenu du registre n° `RD` du coprocesseur 0 dans le registre GPR n° `RT`

Par exemple:
`mtc0 $5, $14` est codé avec `0x40857000`

En effet : `OPCOD` = `010000"` , le bit `INS 25` est à `0` et le bit `INS 23` est à `1` , donc :
— `mtc0 $5, $14` = `0b 010000 | 0.1.. | $5 | $14 | ..... | ......`
— `mtc0 $5, $14` = `0b 010000 | 0.1.. | 00101 | 01110 | ..... | ......`
— `.` est un joker qui peut être remplacé par `0` ou `1` , utilisons `0`
— `mtc0 $5, $14` = `0b 010000 | 00100 | 00101 | 01110 | 00000 | 000000`
— `mtc0 $5, $14` = `0b 0100 0000 1000 0101 0111 0000 0000 0000`
— `mtc0 $5, $14` = `0x 40857000`

5. Cause d'entrée et de sortie du noyau du système d'exploitation

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 le signal RESET.

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

5.A. Entrée pour cause d'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, afin que l'instruction fautive ne modifie pas la valeur d'un registre visible ou 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 (comme un `lw` à une adresse non multiple de 4) ou alors une adresse se trouvant dans la partie *kernel* alors que le processeur est en mode *user*.

ADES
Adresse illégale en écriture : adresse non alignée (comme un `sw` à une adresse non multiple de 4) ou alors un accès à une donnée dans la partie *kernel* alors que le processeur est en mode *user*.

DBE
Data Bus Error : le système mémoire signale une erreur en activant le signal `BERR` (Bus ERRor) 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 Error : 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 *user*.

Dans tous les cas, le processeur doit passer en mode *kernel* et se brancher au noyau du système d'exploitation implanté à l'adresse `0x80000180`. De manière plus détaillée, lorsque le processeur détecte une exception, il réalise les opérations suivantes :

- sauvegarde du registre `PC` (l'adresse de l'instruction fautive) dans le registre `C0_EPC` ;
- passage en mode *kernel* et masquage les interruptions en mettant `1` dans le bit EXL de `C0_SR` ;
- sauvegarde éventuelle de l'adresse fautive dans `C0_BAR` ;
- écriture du type de l'exception dans le registre `C0_CAUSE` ;
- branchement à l'adresse `0x80000180`.

Le fonctionnement des registres de cause (`C0_CAUSE`) et de status (`C0_SR`) est détaillé dans les sections 6. et 7. de ce document. Pour information, après avoir identifié que la cause d'entrée est une exception (en examinant le contenu du registre `C0_CAUSE`), le noyau se branche 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.

5.B. Entrée pour cause d'interruptions

Dans un ordinateur, nous avons vu qu'il y a au moins un processeur, une mémoire et des contrôleurs de périphériques. Les périphériques permettent, par exemple, de communiquer avec le monde extérieur (par exemple le terminal texte). Les périphériques reçoivent des commandes dans leurs registres par des instructions de lecture et d'écriture (`lw` / `sw`) venant du processeur.

Lorsqu'ils ont terminé une commande ou lorsqu'ils ont reçu ou calculé des données, les contrôleurs de périphériques peuvent le signaler au processeur par des **requêtes d'interruption** (IRQ pour Interrupt Request en anglais). **Une requête d'interruption est un signal d'état** produit par un contrôleur de périphérique avec deux états possibles : **actif** (ou levé) qui signifie que le contrôleur demande que le noyau intervienne ou **inactif** (ou baissé) qui signifie que le contrôleur n'a pas de demande. Les requêtes d'interruptions sont donc des notifications de fins de commandes ou d'arrivée de données sur un canal d'entrée ou encore des ticks d'horloge.

Les requêtes d'interruption sont envoyées sur des *lignes d'interruption*. **Une *ligne d'interruption* est un fil électrique qui relie un contrôleur de périphérique au processeur** et qui peut prendre les deux états des requêtes : actif/inactif (ou levé/baissé).

Le processeur MIPS32 possède 6 entrées de lignes d'interruptions externes qui peuvent être *masquées* globalement ou individuellement. Nous n'utiliserons qu'une seule de ces 6 entrées dans le prototype des TP.

Quand le noyau décide *masquer une interruption*, cela signifie qu'il décide ne pas tenir compte de l'état de la ligne d'interruption. Les interruptions peuvent être masquées à certains moments pendant un temps borné lorsque le noyau est en train de faire des opérations critiques qui doivent être réalisées de manière atomique (sans être interrompu). Si une ligne d'interruption s'active alors qu'elle est masquée, alors le processeur ne la verra et donc le noyau ne la traitera qu'au moment où la ligne sera démasquée.

Si elles ne sont pas masquées alors elles sont prises en compte à la fin de l'exécution de l'instruction en cours. Une requête émise par un contrôleur de périphérique doit être maintenue active par le contrôleur tant qu'elle n'a pas été prise en compte par le noyau du système d'exploitation.

Même si l'activation d'une ligne d'interruption est toujours un évènement attendu par le noyau du système d'exploitation, elle survient de manière asynchrone par rapport au programme en cours d'exécution.

Le processeur doit alors passer alors en mode système et se brancher au noyau du système d'exploitation. De manière plus détaillée, lorsque le processeur reçoit une interruption (c'est-à-dire qu'une de ces lignes en entrée est active et non masquée), alors il réalise les opérations suivantes :

- sauvegarde de PC+4 (l'adresse de retour) dans le registre `C0_EPC` ;
- passage en mode *kernel* et masquage des interruptions dans `C0_SR` : `C0_SR`.`EXL` ← `1` ;
- écriture qu'il s'agit d'une interruption dans le registre `C0_CAUSE` ;
- branchement à l'adresse `0x80000180`.

Pour information, 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*).

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.

5.C. Entrée pour cause d'appels système

L'instruction `syscall` permets 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. Quand le processeur exécute l'instruction `syscall, il réalise les opérations suivantes :

- sauvegarde du `PC` (l'adresse de l'instruction) dans le registre `C0_EPC` (l'adresse de retour est `PC` + `4`) ;
- passage en mode *kernel* et masquage des interruptions dans `C0_SR` : `C0_SR`.`EXL` ← `1` ;
- écriture de la cause du déroutement dans le registre `C0_CAUSE` (ici `c0_cause`.`code` ← `8`);
- branchement à l'adresse `0x80000180`.

Pour information, 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.

5.D. Entrée pour cause de 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 *normalement* charger le code du noyau du système d'exploitation dans la mémoire depuis le disque ou le réseau, puis se brancher à la fonction d'initialisation du noyau. Cette dernière 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 préchargé en mémoire et le code de boot se contente d'appeler la fonction d'initialisation après avoir juste initialisé le pointeur de pile `$29`.

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".

5.E. Sortie du noyau

Pour reprendre l'exécution de l'application qui a effectué un appel système (instructions `syscall`) ou alors qui a été interrompue par une requête d'interruption, il faut exécuter l'instruction `eret`. Cette instruction modifie le contenu du registre `C0_SR` en mettant `0` dans son bit `EXL`, et effectue un branchement à l'adresse contenue dans le registre `C0_EPC`. Le fonctionnement du registre `C0_SR` détaillé dans la section **6**.

6. Fonctionnement du registre d'état `c0_sr`

Le registre `C0_SR` contient l'état du processeur. Il définit le comportement du processeur vis-à-vis des requêtes d'interruptions, c'est-à-dire que c'est lui qui contient les masques des lignes d'interruptions matérielles et logicielles, et il définit le mode d'exécution, mode *kernel* ou en mode *user*.

La figure suivante présente le contenu des 16 bits de poids faible du registre `C0_SR`. Dans cette version du MIPS32, nous n'utilisons que 12 bits:

IM[7:0]								0	0	0	UM	0	ERL	EXL	IE
15							8	7	6	5	4	3	2	1	0

`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 masquées
`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 sur l'état du processeur :
  a. 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`.
  b. 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`.
  c. Les trois types d'évènements qui déclenchent le branchement au noyau: (interruptions, exceptions et appels système) mettent le bit `EXL` à `1`, ce qui donc masque les interruptions et autorise l'accès aux ressources protégées.
  d. 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.
  e. L'instruction `eret` force le bit `EXL` à `0`, et l'état du processeur est alors défini par par les valeurs des bits `UM` et `IE`.
  f. Pour exécuter un programme utilisateur en mode protégé, avec interruptions activées, le registre `C0_SR` doit contenir la valeur `0xFF11` (c'est-à-dire `IM[7:0] = 0xFF` ; `UM = 1` ; `IE = 1`). Par conséquent avant d'exécuter l'instruction `eret`, le noyau devra avoir écrit `0xFF13` dans `C0_SR` (c'est-à-dire `IM[7:0] = 0xFF` ; `UM = 1` ; `IE = 1` ; `EXL = 1`) et le noyau doit aussi mettre l'adresse de l'instruction utilisateur dans `C0_EPC`.

- Lors de l'activation du RESET :
  a. `C0_SR` contient donc la valeur `0x0004` (`0b0000000000000100` donc `ERL = 1`) .

7. Fonctionnement 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 du noyau. Les 2 bits `SWI[1:0]` représentent les requêtes d'interruption logicielle.

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

IRQ[5:0]						SWI[1:0]		0	0	XCODE[3:0]				0	0
15					10	9	8	7	6	5	4	3	2	1	0

- 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