Exceptions et Entrées/Sorties

Mécanismes d'exceptions dans MIPS

Les branchements et les sauts changent le flux d'exécution dans in programme. Les exceptions modifient également le flux d'exécution. L'architecture MIPS appelle une **exception** tout changement **inattendu** dans le flux de d'exécution dans un programme, quelle que soit sa source.

Une exception est dite **synchrone** si elle est provoquée par une instruction du programme en cours. Les exceptions arithmétiques, les adresses mémoire non valides générées par les instructions de chargement et de stockage et les instructions d'interruption sont des exemples d'exceptions synchrones.

Une exception est dite **asynchrone** si elle est provoquée par un périphérique d'E/S. Ceci est également appelé une **interruption** matérielle, qui n'est pas liée à l'exécution du programme. Les interruptions peuvent être provoquées par divers périphériques d'E/S, tels que le clavier, le minuteur et le contrôleur de disque.

Lorsqu'une exception se produit, le contrôle est transféré à un **gestionnaire d'exceptions**, écrit spécifiquement dans le but de gérer les exceptions. Après avoir exécuté le gestionnaire d'exceptions, le contrôle est renvoyé au programme. Le programme continue comme si de rien n'était. Le gestionnaire d'exceptions apparaît sous la forme d'une procédure appelée soudainement dans le programme, sans paramètre ni valeur de retour.

Le processeur MIPS fonctionne en **mode utilisateur** ou en **mode superviseur** (*kernel* en anglais). Les programmes utilisateur (applications) s'exécutent en mode utilisateur. La CPU entre en mode superviseur lorsqu'une exception se produit. Le mécanisme de traitement des exceptions est implémenté par le **Coprocesseur 0**, qui possède plusieurs registres importants, tels que : **vaddr**, **status**, **cause** et **epc**, qui enregistrent des informations sur l'exception.

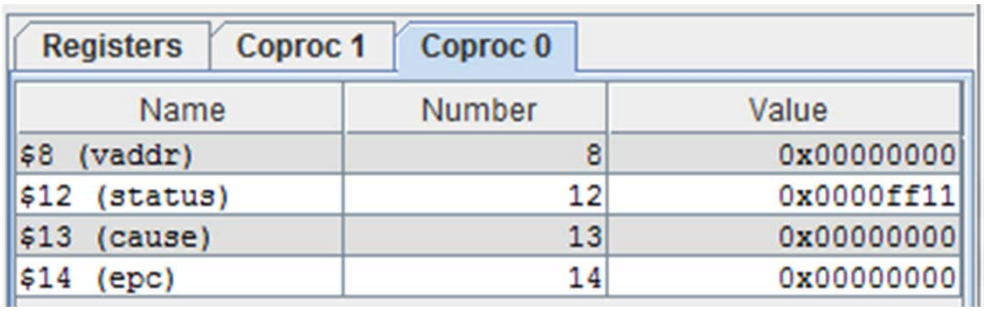
vaddr ($8) : Contient l'adresse de mémoire non valide provoquée par une instruction de chargement (load), le stockage (store) ou la récupération (fetch).

status ($12) : Contient le masque d'interruption global et les bits d'activation sélective des interruptions (voir ci-dessous).

cause ($13) : Contient le type d'exception et le masque de bits associé avec les interruptions en attente de traitement (voir ci-dessous).

epc ($14) : Contient l'adresse de l'instruction lorsque l'exception s'est produite.

Le simulateur MARS affiche les valeurs des registres : **vaddr**, **status**, **cause** et **epc** sous l'onglet Coprocessor 0, comme illustré à la figure 1.



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$8 (vaddr)	8	0x00000000
$12 (status)	12	0x0000ff11
$13 (cause)	13	0x00000000
$14 (epc)	14	0x00000000

Figure 1. Les registres du Coprocesseur 0 dans MARS

Des exemples d'exceptions sont illustrés à la figure 2. Le premier exemple initialise le registre **$t0** avec **0x7fffffff** et **$t1** avec **1**. L'instruction **addu** ignore le débordement (overflow). Cependant, l'instruction **add** détecte et provoque une exception de débordement arithmétique. Dans le deuxième cas, un exemple de stockage de données à une adresse illégale en mémoire est donné. Le troisième exemple concerne le chargement d'un mot d'une adresse mal alignée en mémoire. Le dernier exemple illustre l'insertion d'un point d'arrêt (instruction **break**) dans le programme. Tous ces exemples provoquent des exceptions et nécessitent un traitement spécial.

Exemples d'exceptions	Registres du Coprocessor 0
# Exception de débordement arithmétique **li $t0, 0x7fffffff # $t0 = MAX_INT** **li $t1, 1 # $t1 = 1** **addu $t2, $t0, $t1 # Ignore Overflow** **add $t3, $t0, $t1 # Detect Overflow**	Registers | Coproc 1 | Coproc 0 Name / Number / Value $8 (vaddr) 8 0x00000000 $12 (status) 12 0x0000ff13 $13 (cause) 13 0x00000030 $14 (epc) 14 0x00400010
# Exception d'addresse de stockage **# Ne peut pas stocker à l'adresse 4** **li $t0, 4** **li $a0, 5** **sw $a0, ($t0)**	Registers | Coproc 1 | Coproc 0 Name / Number / Value $8 (vaddr) 8 0x00000004 $12 (status) 12 0x0000ff13 $13 (cause) 13 0x00000014 $14 (epc) 14 0x0040001c
# Exception d'adresse de lecture mal alignée **.data** **arr: .word 12, 17** **. . .** **la $t0, arr** **lw $t0, 1($t0)**	Registers | Coproc 1 | Coproc 0 Name / Number / Value $8 (vaddr) 8 0x10010001 $12 (status) 12 0x0000ff13 $13 (cause) 13 0x00000010 $14 (epc) 14 0x00400028
# Exception point d'arrêt (Breakpoint) **# Causé par l'instruction break** **.text** **. . .** **break**	Registers | Coproc 1 | Coproc 0 Name / Number / Value $8 (vaddr) 8 0x00000000 $12 (status) 12 0x0000ff13 $13 (cause) 13 0x00000024 $14 (epc) 14 0x0040002c

Figure 2. Exemples d'exceptions et valeurs correspondantes des registres du coprocesseur 0

La deuxième colonne de la figure 2 montre les registres du coprocesseur 0 lorsqu'une exception se produit. Le registre de programme d'exception **$14** (**epc**) stocke l'adresse de l'instruction à l'origine de l'exception. La valeur de **epc** dans la figure 2 est **0x00400010**, qui correspond à l'adresse de l'instruction **add** qui a provoqué le débordement arithmétique. Dans l'exemple suivant, le registre **epc** contient la valeur **0x0040001c** qui correspond à l'adresse de **sw** ayant tenté d'écrire dans une adresse illégale en mémoire. Dans le troisième exemple, ce registre contient la valeur **0x00400028**, qui est l'adresse de **lw** qui a généré une adresse mal alignée en mémoire.

Le registre **$13** (**cause**) fournit des informations sur la cause d'une exception (**code d'exception**) et sur les interruptions en attente, le cas échéant. Le code d'exception est stocké dans les bits 2 à 6 du registre des causes. Les champs de bits du registre de cause sont illustrés à la figure 3.

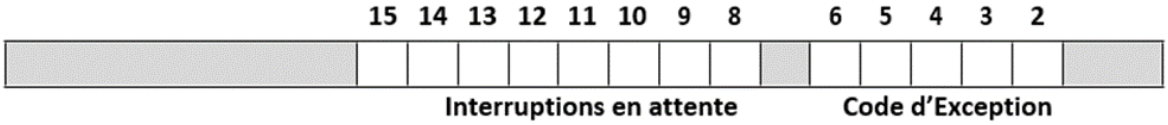


Figure 3. Le registre **cause $13**

L'architecture MIPS définit des codes numériques (valeurs) pour différents types d'exceptions. Certains de ces codes sont énumérés à la figure 4. L'outil MARS simule certains de ces codes d'exception.

Code	Nom	Description
0	INT	Interruption matérielle
4	ADDRL	Exception d'erreur d'adresse provoquée par le chargement de donnée (load) ou la récupération d'instruction (fetch)
5	ADDRS	Exception d'erreur d'adresse provoquée par l'instruction de sauvegarde de donnée (store).
6	IBUS	Erreur de bus lors de la récupération d'une instruction
7	DBUS	Erreur de bus lors du chargement ou de sauvegarde de données
8	SYSCALL	Exception d'appel système provoquée par l'instruction **syscall**
9	BKPT	Exception de point d'arrêt causée par l'instruction **break**
10	RI	Exception d'instruction réservée
12	OVF	Exception de débordement arithmétique
13	TRAP	Exception provoquée par une instruction **trap**
15	FPE	Exception provoquée par une instruction en virgule flottante

Figure 4 Quelques codes d'exception dans MIPS

Si une instruction *load*, *store*, *jump* ou *branch* crée une exception, alors le registre **$8** (**vaddr**) contient l'adresse de mémoire non valide qui a provoqué l'exception. Reprenons les exemples de la figure 2, l'adresse de données non autorisée de **sw** qui a provoqué l'exception est **vaddr** = **0x00000004**, et l'adresse de données mal alignée de **lw** qui a provoqué l'exception est **vaddr** = **0x10010001**.

Le registre **$12** (**statut**) est montré à la figure 5. Le bit 0 est l'activation d'interruption (**IE**), qui active ou désactive les interruptions. Le bit 1 est appelé niveau d'exception (Exception Level) (**EL**). Ce bit est normalement à 0 sauf si une exception se produit, dans ce cas il est mis à 1.

Le champ de **masque d'interruption** contient 8 bits. Un bit pour chacun des six niveaux d'interruption du matériel et deux du logiciel. Quand un bit de masque est égal à 1, les interruptions de ce niveau sont autorisées pour interrompre le processeur. Quand un bit de masque est égal à 0, cela désactive les interruptions associées à ce niveau. Lorsqu'une interruption se produit, le bit associé dans le registre **cause** (interruption en attente) est activé même si le bit de masque est désactivé. Lorsqu'une interruption est en attente, elle sera traitée par le processeur une fois son bit de masque est activé.

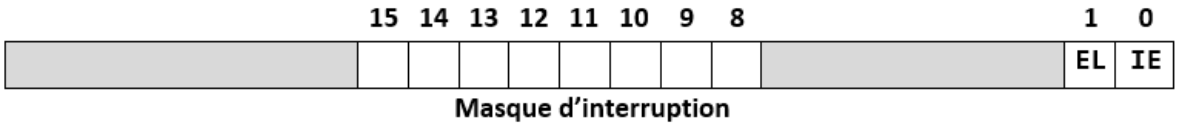


Figure 5. Le registre **Status $12**

Instructions du Coprocesseur 0

L'architecture MIPS définit des instructions spéciales qui provoquent des exceptions et font passer le processeur du mode utilisateur au mode superviseur. Ce sont les instructions **trap**, **break** et **syscall** :

Instruction	Description
teq rs, rt	Génère une exception **trap** si le registre **rs** est égal au registre **rt**
tne rs, rt	Génère une exception **trap** si le registre **rs** n'est pas égal au registre **rt**
tlt rs, rt	Génère une exception **trap** si le registre **rs** est inférieur au registre **rt**
. . .	Il existe d'autres instructions de **trap** non listées ici.
break	Génère une exception de breakpoint.
syscall	Gènère une exception d'appel système. Le numéro de service est spécifié dans **$v0.**

Les instructions de déroutement (**teq**, **tne**,… etc.) génèrent l'exception TRAP (code 13). L'instruction **break** génère l'exception de point d'arrêt (code 9). Cependant, le simulateur MARS fournit des services pour l'instruction **syscall** sans déclencher d'exception. Sur un système réel, l'instruction **syscall** déclenche l'exception d'appel système (code 8), qui est traitée par le système d'exploitation.

Lorsqu'une exception se produit, le processeur bascule en mode superviseur (kernel en anglais). Les registres du coprocesseur 0 ne sont accessibles que lorsque le processeur traite une exception en mode superviseur.

Les contenus des registres peuvent être transférées du/vers le coprocesseur 0 à l'aide des instructions citées ci-dessous. Les instructions de lecture et de stockage transférant des données entre les registres du coprocesseur 0 et la mémoire sont également indiquées. Ces instructions peuvent être utilisées lors de l'écriture d'un **gestionnaire d'exceptions**.

Instruction	Description
mfc0 rd, C0src	Copie du registre **C0src** du coprocesseur 0 dans le registre destination **rd**.
mtc0 rs, C0dst	Copie du registre source **rs** vers le registre **C0src** du coprocesseur 0.
lwc0 C0dst, addr	Lit un mot de la mémoire et le copie dans le registre **C0dst** du coprocesseur 0.
swc0 C0src, addr	Sauvegarde le contenu du registre **C0src** du coprocesseur 0 dans la mémoire.
eret	Reset **EL = 0** (activation du mode utilisateur) et retour : **$pc** = **$epc**

Routines d'exception

La structure d'un programme MIPS est illustrée à la figure 6. Le système d'exploitation apparaît dans la moitié supérieure de l'espace d'adressage, accessible uniquement lorsque le processeur s'exécute en mode superviseur. Le segment de code du superviseur du système d'exploitation commence à l'adresse **0x80000000** et le segment de données du superviseur commence à l'adresse **0x90000000**. Le dernier segment de l'espace d'adressage est *mappé* aux périphériques d'E/S à partir de l'adresse **0xffff0000**. Ceci est connu sous le nom d'E/S mappées en mémoire (Memory-Mapped I/O).

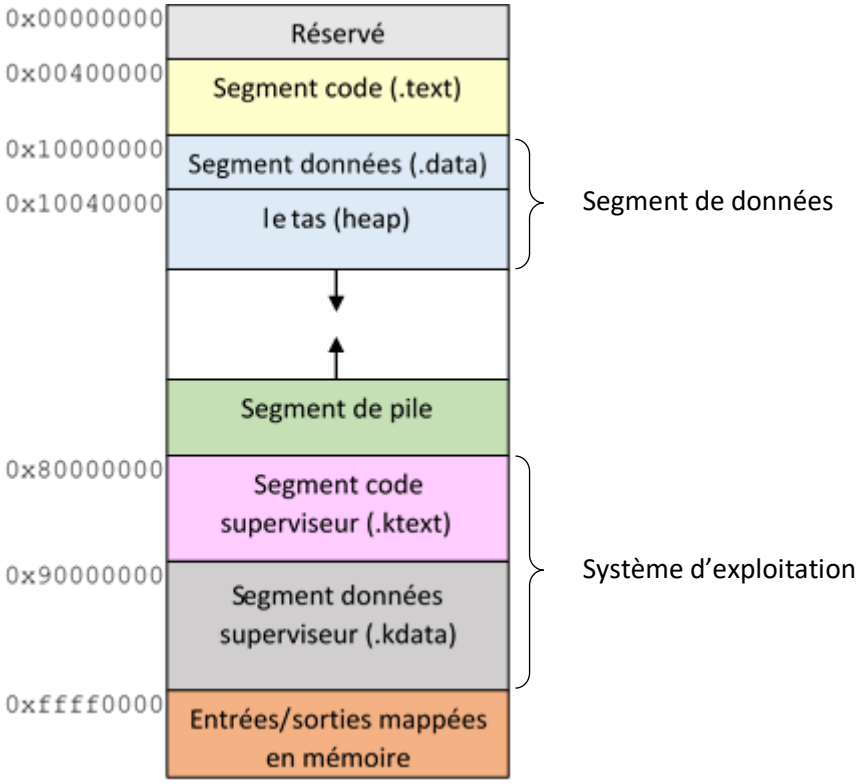


Figure 6. Structure d'un programme MIPS en mémoire.

Lorsqu'une fonction est appelée à l'aide de l'instruction **jal**, le contrôle est transféré à l'adresse fournie par l'instruction et l'adresse de retour est enregistrée dans le registre **$ra**. Dans le cas d'une exception, il n'y a pas d'appel explicite. Dans MIPS, lorsqu'une exception se produit, le contrôle est transféré à l'adresse fixe **0x80000180**. Le gestionnaire d'exception doit être implémenté à cette adresse.

L'adresse de retour de l'exception ne peut pas être enregistrée dans **$ra** car cela modifiera une adresse de retour qui a été placée dans ce registre avant l'exception. Le registre **epc** (Exception Program Counter) est utilisé pour stocker l'adresse de l'instruction qui était en cours d'exécution lors de la génération de l'exception.

Un gestionnaire d'exceptions peut être écrit dans le même fichier que le programme normal ou dans un fichier séparé. Le gestionnaire d'exceptions doit commencer à l'adresse fixe **0x80000180**. Cette adresse est dans le segment de texte du superviseur. S'il n'y a pas d'instruction à l'adresse **0x80000180**, MARS mettra fin au programme MIPS avec un message d'erreur approprié. Un exemple de gestionnaire d'exception qui affiche l'adresse de l'instruction à l'origine de l'exception et son code est présenté ci-dessous :

```
# Exception Handler starts here
.ktext 0x80000180
move $k0, $at                    # $k0 = $at
la $k1, _regs                    # $k1 = address of _regs
sw $k0, 0($k1)                   # save $at
sw $v0, 4($k1)                   # save $v0
sw $a0, 8($k1)                   # save $a0

la $a0, _msg1                    # $a0 = address of _msg1
li $v0, 4                        # $v0 = service 4
syscall                          # Print _msg1
mfc0 $a0, $14                    # $a0 = EPC
li $v0, 34                       # $v0 = service 34
syscall                          # print EPC in hexadecimal

la $a0, _msg2                    # $a0 = address of _msg2
li $v0, 4                        # $v0 = service 4
syscall                          # Print _msg2
mfc0 $a0, $13                    # $a0 = cause
srl $a0, $a0, 2                  # shift right by 2 bits
andi $a0, $a0, 31                # $a0 = exception code
li $v0, 1                        # $v0 = service 1
syscall                          # Print exception code

la $a0, _msg3                    # $a0 = address of _msg3
li $v0, 4                        # $v0 = service 4
syscall                          # Print _msg3

la $k1, _regs                    # $k1 = address of _regs
lw $at, 0($k1)                   # restore $at
lw $v0, 4($k1)                   # restore $v0
lw $a0, 8($k1)                   # restore $a0

mtc0 $zero, $8                   # clear vaddr
mfc0 $k0, $14                    # $k0 = EPC
addiu $k0, $k0, 4                # Increment $k0 by 4
mtc0 $k0, $14                    # EPC = point to next instruction

eret                             # exception return: PC = EPC
```

```
# kernel data is stored here
.kdata

_msg1:   .asciiz    "\nException caused by instruction at address: "
_msg2:   .asciiz    "\nException Code = "
_msg3:   .asciiz    "\nIgnore and continue program ...\n"
_regs:   .word 0:3              # Space for saved registers
```

Figure 7. Exemple d'une routine de gestion d'exception

L'écriture d'un gestionnaire d'exceptions n'est pas une tâche facile. Les étapes à effectuer sont les suivantes :

1. Enregistrer les registres avant de les modifier dans le gestionnaire d'exceptions.
2. Lire les registres du coprocesseur 0 pour déterminer quelle exception s'est produite.
3. Exécuter le gestionnaire spécifique pour le code d'exception détecté, généralement via une table de branchement.
4. Restaurer tous les registres modifiés par le gestionnaire d'exceptions.
5. Poursuivre l'exécution du programme en mode utilisateur ou terminer le programme s'il ne peut pas être redémarré.

Le gestionnaire d'exceptions doit conserver la valeur de tout registre qu'il pourrait modifier, afin que l'exécution du programme interrompu puisse se poursuivre ultérieurement. L'architecture MIPS réserve les registres **$26** et **$27** (**$k0** et **$k1**) pour l'utilisation du gestionnaire d'exceptions. Le gestionnaire peut modifier ces deux registres sans avoir à les sauvegarder au préalable. Les autres registres doivent être enregistrés en mémoire avant de pouvoir être modifiés par le gestionnaire d'exceptions. Par exemple, le gestionnaire d'exceptions de la figure 7 enregistre les registres **$at**, **$a0** et **$v0** dans le segment de données du superviseur.

Le registre **epc** contient l'adresse de l'instruction au moment où l'exception s'est produite. L'adresse de retour doit être incrémentée de 4 pour éviter d'exécuter à nouveau la même instruction. La séquence de retour d'une exception logicielle peut être écrite comme suit :

```
mfc0  $k0, $14     #  $k0 = epc
addiu $k0, $k0, 4  #  increment $k0 by 4
mtc0  $k0, $14     #  epc = point to next instruction
eret               #  exception return: pc = epc
```

Dans MARS, plutôt que d'écrire un gestionnaire d'exceptions à la fin de chaque programme MIPS, il est préférable d'écrire le gestionnaire d'exceptions dans un fichier séparé, puis d'ouvrir la boîte de dialogue «Exception Handler …» dans les paramètres MARS et d'inclure le fichier du gestionnaire d'exceptions dans tous vos programmes MIPS.

Entrées/sorties mappées en mémoire

Dans tout ordinateur, les périphériques d'entrée et de sortie se trouvent en dehors de la puce du processeur. Un processeur MIPS communique avec les périphériques d'E/S à l'aide d'une technique appelée E/S mappée en mémoire (**memory-mapped I/O**).

Avec les E/S mappées en mémoire, il n'est pas nécessaire d'ajouter des instructions supplémentaires au jeu d'instructions MIPS. Toute instruction **lw** ou **sw** avec une adresse effective égale ou supérieure à **0xffff0000** n'accédera pas à la mémoire principale. Ces adresses sont réservées pour permettre l'accès aux registres des périphériques d'E/S. Les contrôleurs de périphérique d'E/S doivent être connectés au bus d'E/S du système, comme illustré à la figure 8.

Une logique de décodage d'adresse unique est associée à chaque registre d'E/S. Lorsque le processeur MIPS lit ou écrit dans l'une de ces adresses, il lit ou écrit dans un registre sélectionné de l'un des contrôleurs de périphérique d'E/S. Le processeur peut lire des données sur l'état du périphérique d'E/S et écrire des données de contrôle pour modifier l'état du périphérique.

Les deux registres associés au clavier sont les registres de contrôle et de données du récepteur. Ceux-ci sont mappés aux adresses **0xffff0000** et **0xffff0004** respectivement. Pour communiquer avec le clavier, le processeur lit le registre de contrôle. Tant que le **bit prêt** (**ready bit**) est à zéro, le processeur continue à lire le registre de contrôle dans une boucle. Cette approche est appelée interrogation (**polling**).

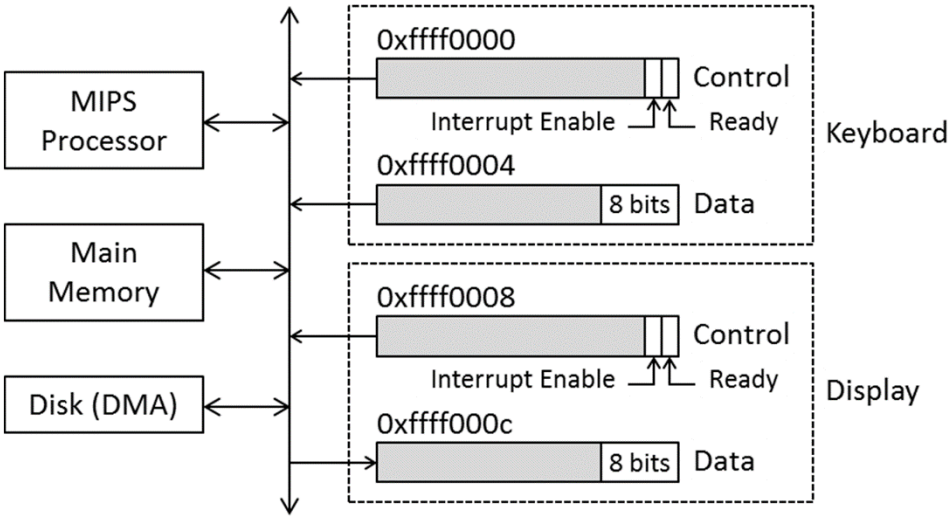


Figure 8. BUS E/S pour un système MIPS

Lorsqu'une touche est enfoncée, le registre de données enregistre le caractère associé et le **bit prêt** (**ready**) est activé. Ensuite, le processeur lit le caractère. Le code MIPS suivant fournit un exemple d'accès mappé en mémoire aux registres de contrôle du clavier et de données via une interrogation (polling) :

```
li     $t0, 0xffff0000      # Address of keyboard control register
li     $t1, 0               # Initialize wait_counter = 0

wait_keyboard:

lw     $t2, ($t0)           # Read the keyboard control register
andi   $t2, $t2, 1          # Extract ready bit
addiu  $t1, $t1, 1          # wait_counter++ (counts iterations)
beqz   $t2, waitkeyboard    # loop back while not ready
lw     $a0, 4($t0)          # Get character from keyboard
```

La fréquence de saisie des caractères sur le clavier est très lente par rapport à la vitesse à laquelle le processeur MIPS peut exécuter des instructions. En règle générale, des millions d'instructions sont exécutées jusqu'à ce qu'une touche soit enfoncée. Dans notre exemple, le registre **$t1** garde une trace du nombre d'itérations dans la boucle **wait_keyboard**. L'instruction **lw** en dehors de la boucle extrait le caractère depuis le registre de données du clavier, ce qui efface également le **bit prêt**. Un programme MIPS peut seulement lire le contenu du registre de données du clavier. L'écriture dans ce registre n'a aucun effet.

La communication avec la carte graphique peut être effectuée de manière similaire via une interrogation. Les registres d'affichage sont mappés aux adresses **0xffff0008** et **0xffff000c**. Tant que le **bit prêt** est à zéro, le processeur continue à lire le registre de contrôle en boucle. Nous ne devons pas écrire de caractère dans le registre de données tant que l'affichage n'est pas prêt à le recevoir. Le contrôleur d'affichage efface le **bit prêt** lorsqu'un caractère est écrit dans le registre de données. Il remet à nouveau le bit prêt à 1 après que le caractère soit affiché et il est prêt à recevoir le prochain caractère.

```
li    $t0, 0xffff0008      # Address of display control register

wait_display:

lw    $t2, ($t0)           # Read the display control register
andi  $t2, $t2, 1          # Extract ready bit
beqz  $t2, wait_display    # loop back while not ready
sw    $a0, 4($t0)          # Send character to display
```

Le simulateur MARS fournit un outil pour simuler le clavier et l'affichage, comme illustré à la figure 9. Appuyez sur "Connect to MIPS" pour connecter cet outil au programme MIPS. Vous devez activer cet outil pour communiquer caractère-par-caractère avec le clavier et l'écran. Le code qui communique avec les périphériques d'E/S à ce niveau est appelé **pilote de périphérique**.

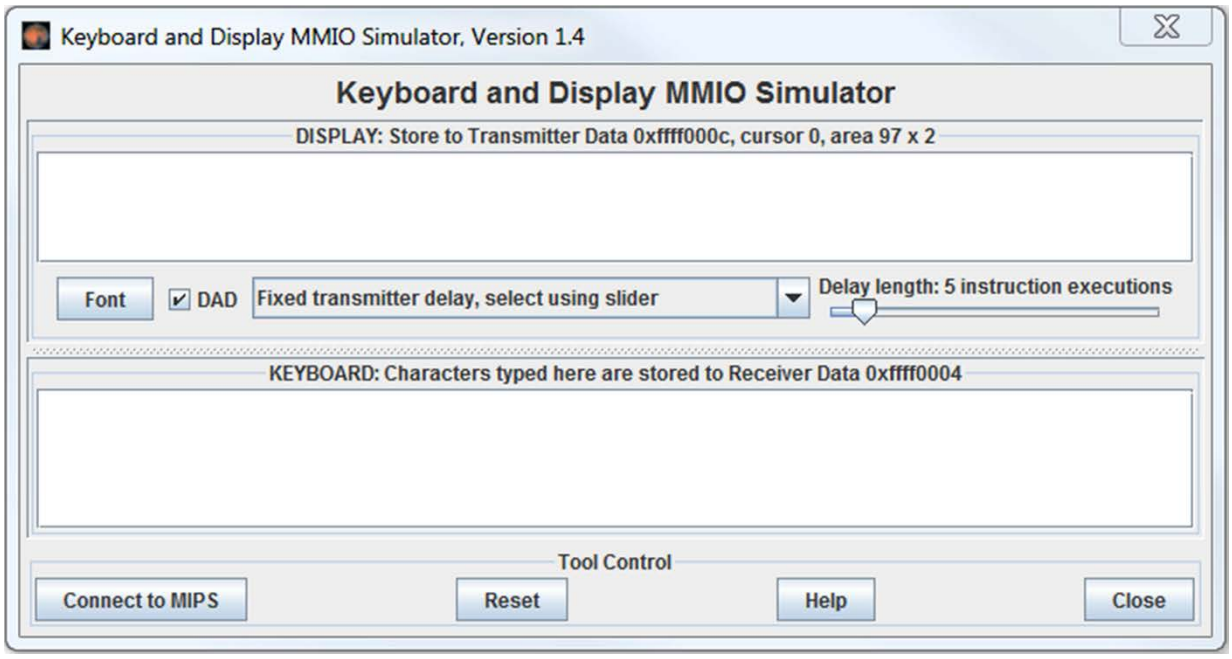


Figure 9. Simulateur MMIO dans MARS pour le clavier et l'affichage

L'inconvénient principal de la méthode du sondage polling) est qu'elle maintient le processeur occupé, gaspillant des millions de cycles avant que le périphérique d'E/S (tel que le clavier) ne soit prêt. Une alternative à l'interrogation consiste à utiliser des interruptions. Les interruptions peuvent être activées pour le clavier en définissant le bit **Interrupt Enable** dans le registre de contrôle du clavier comme suit :

```
li  $t0, 0xffff0000   # Address of keyboard control register
li  $t1, 2
sw  $t1, ($t0)        # Enable keyboard interrupt
```

Lorsqu'une touche est enfoncée, le clavier envoie un signal d'interruption au processeur MIPS et définit le registre de **cause $13**sur la valeur **0x00000100**. Le bit 8 dans le registre de cause est défini pour indiquer que le clavier a interrompu le processeur. Un gestionnaire d'interruption doit être écrit pour lire le caractère sur le clavier et renvoyer sa valeur au programme en cours d'exécution. Cela nécessite de modifier le code du gestionnaire d'exceptions illustré à la figure 7 pour gérer les interruptions.

Les contrôleurs de périphérique d'E/S disque et Ethernet utilisent un accès DMA (Direct Memory Access) pour transférer des blocs de données directement entre le périphérique et la mémoire. À mesure que les contrôleurs deviennent plus complexes, plus de deux registres sont associés à chaque contrôleur de périphérique. Ces appareils ne font pas partie du simulateur MARS. En général, le fournisseur de tout périphérique d’E/S doit fournir aux programmeurs une explication sur la manière de communiquer correctement avec le contrôleur de périphérique d’E/S.