Compte-rendu TP n°2, MULTI, 2019-2020

Aymeric Agon-Rambosson

Mercredi 12 février 2020

Cette semaine, on va remplacer le processeur câblé de la dernière fois par un modèle programmable.

Question C1

On a direct mapping, soit icache_ways = 1

On a 16 octets par ligne de cache, soit 4 mots, donc icache_words = 4

1024 / 16, soient 64 lignes, donc icache_sets = 64

De même pour les caches de données.

Question C2

Il faut pouvoir garantir que la machine rebootera après un arrêt intempestif.

Pour que notre automate soit déterministe, il faut qu'il ait un unique état initial connu et constant.

Cet état initial est donné par le contenu du segment seg_reset.

Il faut absolument que ce segment mappe vers une zone de la mémoire en lecture seule.

Question C3

Le segment seg_tty doit être non cachable, parce que les caractères affichés dans le terminal doivent toujours correspondre effectivement à ce que le processeur connaît.

Le cache permet que ces deux valeurs soient différentes, même transitoirement, et c'est inacceptable.

Question C4

Les segments protégés sont :

- seg_reset
- seg_kcode
- seg_kdata
- seg_kdata
- seg_tty

On les repère à ce qu'ils font partie des adresses hautes (bit de poids fort à 1).

Question D1

Le programme utilisateur doit donner au noyau :

- la référence de l'appel système qu'il appelle
- les paramètres

La première information est facile à transmettre :
Les noms et les comportements des appels systèmes sont réputés connus de l'utilisateur (ils sont documentés) : il suffit de passer un nom, et le noyau a une liste statique des appels systèmes. Il cherche dans sa liste et trouve (ou non, auquel cas il rend une erreur).

La deuxième information est un peu plus délicate à transmettre : le noyau et le userspace n'ont pas la même pile : le noyau, qui a tous les droits, peut aller chercher les arguments directement dans la pile utilisateur du processus qui a fait l'appel système (c'est ce que faisait unix v5).

Sinon, comme c'est le cas ici, on peut stocker les arguments dans des registres visibles du processeur. Cette technique suppose une quantité limitée d'arguments.

En général, c'est même une technique mixte qui est utilisée : les premiers arguments peuvent être mis dans des registres, et la suite peut-être mise ou bien dans la pile ou bien dans une zone précise de l'espace mémoire du processus appelant.

Question D2

Le tableau _cause_vector contient les raisons d'entrée en mode noyau. Il est initialisé dans le fichier exc_handler.c.

Le tableau _syscall_vector contient les points d'entrée des handlers des appels systèmes. Il est initialisé dans le fichier sys_handler.c.

D'après le code assembleur de la fonction _sys_handler, les handlers des appels systèmes sont indexés par leurs 5 bits de poids faible (donc 32 handlers) :

```
andi        $26,$2,0x1F
```

Question D3

Dans l'ordre :

L'utilisateur appelle la fonction utilisateur proctime().

La fonction proctime() appelle la fonction inlinée sys_call(), avec comme premier argument 0x01, soit l'index de la fonction **noyau** _proctime() dans le tableau _syscall_vector.

La fonction syscall() enregistre dans des registres du processeur les arguments de la fonction, et il appelle l'instruction MIPS "syscall".

On fait l'hypothèse que cette instruction permet de passer en mode noyau (donc de rendre accessible des instructions processeur et des adresses), et de jump à une adresse bien précise. Ce passage en mode noyau se fait au moyen d'une trappe, soit une interruption matérielle.

On fait l'hypothèse, appuyée sur le code, qu'on saute à l'adresse 0x80000000 (en fait 0x80000000 + 0x180), soit l'adresse de la fonction assembleur _giet. Cette fonction assembleur du noyau a pour but de regarder quelle est la cause de l'interruption matérielle. Il se trouve que la cause de l'interruption matérielle courante suppose l'appel à la fonction assembleur _sys_handler.

On saute donc vers la fonction _sys_handler.

La fonction _sys_handler consulte le tableau _syscall_vector, et voit donc que le handler à appeler est la fonction _proctime.

_sys_handler saute donc vers la fonction _proctime (et il prend la peine de désactiver les interruptions : le noyau est **non-préemptif**) :

```
jalr        $3
mtc0        $0,$12
```

(on est en MIPS 32 bits, avec 5 étages de pipeline et un delayed slot : l'instruction mtc0 est exécutée malgré le jump)

La fonction _proctime consiste en une ligne d'assembleur, qui va lire le contenu du registre $9 du coprocesseur 0, censé apparemment contenir le nombre de cycles de processeurs écoulés depuis le démarrage de la machine (on ne sait pas trop comment, le code ne permet pas de le dire).

Cette valeur est copiée dans une variable de la pile, appelée ret. Cette variable est retournée par la fonction.

On se retrouve à la ligne suivante :

```
lw          $26,16($29)
mtc0        $26,$12
```

La fin de la fonction restaure les pointeurs de pile et d'instructions.

On se retrouve en mode utilisateur, à la fin de la fonction sys_call().

Qui retourne la valeur qui a été opportunément placée dans la pile au bon endroit (dans la variable reg_no_and_output).

On se trouve à la fin de la fonction proctime().

Question D4

Cet appel système dont on vient de détailler le déroulement a le coût en **instructions** suivant, calculé sur la base des fichiers app.bin.txt et sys.bin.txt, qui sont les codes objets désassemblés :

Dans la fonction proctime :

400094: 27bdffe0 addiu sp,sp,-32
400098: afbf001c sw ra,28(sp)
40009c: afbe0018 sw s8,24(sp)
4000a0: 03a0f025 move s8,sp
4000a4: afa00010 sw zero,16(sp)
4000a8: 00003825 move a3,zero
4000ac: 00003025 move a2,zero
4000b0: 00002825 move a1,zero
4000b4: 24040001 li a0,1
4000b8: 0c100000 jal 400000 <sys_call>
4000bc: 00000000 nop

(soient 11 instructions)

Dans la fonction sys_call :

400000: 27bdfff8 addiu sp,sp,-8
400004: afbf0004 sw ra,4(sp)
400008: afbe0000 sw s8,0(sp)
40000c: 03a0f025 move s8,sp
400010: afc40008 sw a0,8(s8)
400014: afc5000c sw a1,12(s8)
400018: afc60010 sw a2,16(s8)
40001c: afc70014 sw a3,20(s8)
400020: 8fc20008 lw v0,8(s8)
400024: 8fc4000c lw a0,12(s8)
400028: 8fc50010 lw a1,16(s8)
40002c: 8fc60014 lw a2,20(s8)
400030: 8fc70018 lw a3,24(s8)
400034: 0000000c syscall

(soient 14 instructions)

Dans la fonction _giet :

```
80000180: 401b6800 mfc0 k1,c0_cause
80000184: 3c1a8200 lui k0,0x8200
80000188: 275a00d0 addiu k0,k0,208
8000018c: 337b003c andi k1,k1,0x3c
80000190: 035bd021 addu k0,k0,k1
80000194: 8f5a0000 lw k0,0(k0)
80000198: 03400008 jr k0
8000019c: 00000000 nop
```

(soient 8 instructions)

Dans la fonction _sys_handler :

```
800001a0: 27bdffe8 addiu sp,sp,-24
800001a4: 401a6000 mfc0 k0,c0_status
800001a8: afba0010 sw k0,16(sp)
800001ac: 401b7000 mfc0 k1,c0_epc
800001b0: 277b0004 addiu k1,k1,4
800001b4: afbb0014 sw k1,20(sp)
800001b8: 305a001f andi k0,v0,0x1f
800001bc: 001ad080 sll k0,k0,0x2
800001c0: 3c1b8200 lui k1,0x8200
800001c4: 277b031c addiu k1,k1,796
800001c8: 037ad821 addu k1,k1,k0
800001cc: 8f630000 lw v1,0(k1)
800001d0: 241bffed li k1,-19
800001d4: 401a6000 mfc0 k0,c0_status
800001d8: 035bd024 and k0,k0,k1
800001dc: 0060f809 jalr v1
800001e0: 409a6000 mtc0 k0,c0_status
```

(soient 17 instructions)

Dans la fonction _proctime :

```
80000520: 27bdfff0 addiu sp,sp,-16
80000524: afbe000c sw s8,12(sp)
80000528: 03a0f025 move s8,sp
8000052c: 40024800 mfc0 v0,c0_count
80000530: afc20000 sw v0,0(s8)
80000534: 8fc20000 lw v0,0(s8)
80000538: 03c0e825 move sp,s8
8000053c: 8fbe000c lw s8,12(sp)
80000540: 27bd0010 addiu sp,sp,16
80000544: 03e00008 jr ra
80000548: 00000000 nop
```

(soient 11 instructions)

De retour dans la fonction _sys_handler :

```
800001e4: 40806000 mtc0 zero,c0_status
800001e8: 8fba0010 lw k0,16(sp)
800001ec: 409a6000 mtc0 k0,c0_status
800001f0: 8fba0014 lw k0,20(sp)
800001f4: 409a7000 mtc0 k0,c0_epc
800001f8: 27bd0018 addiu sp,sp,24
800001fc: 42000018 eret
```

(soient 7 instructions)

De retour dans la fonction sys_call (on est donc sorti du mode noyau) :

400038: 03c0e825 move sp,s8
40003c: 8fbf0004 lw ra,4(sp)
400040: 8fbe0000 lw s8,0(sp)
400044: 27bd0008 addiu sp,sp,8
400048: 03e00008 jr ra
40004c: 00000000 nop

(soient 6 instructions)

De retour dans la fonction proctime :

4000c0: 03c0e825 move sp,s8
4000c4: 8fbf001c lw ra,28(sp)
4000c8: 8fbe0018 lw s8,24(sp)
4000cc: 27bd0020 addiu sp,sp,32
4000d0: 03e00008 jr ra
4000d4: 00000000 nop

(soient 6 instructions)

Donc, depuis l'appel à la fonction proctime par le programme applicatif, à la sortie de cette même fonction, on a un coût en instructions :

37 instructions en mode utilisateur
43 instructions en mode système

Soient un grand total de **80 instructions**.

Si on suppose, en raison des delayed slots, cycles de gels, accès mémoire, etc. . . , un CPI de 2, on a donc **160 cycles** dépensés pour cet accès à proctime.

Question E1

En général, le code de boot doit être exécuté en mode système parce que le boot consiste entre autres à charger le code du noyau en mémoire centrale, et en zone noyau de la mémoire centrale. Pour accéder en écriture à cette zone, on doit être en mode noyau.

Dans notre cas, quand bien même on suppose le code du noyau déjà chargé en mémoire au démarrage de la machine, on est quand même censé manipuler des registres, comme le registre SR, qui n'est accessible qu'en mode noyau.

Question E2

L'adresse du point d'entrée du code applicatif doit se trouver au début du segment seg_data_base (soit exactement à l'adresse 0x01000000) :

```
la              $26,seg_data_base
lw              $26,0($26)          # get the user code entry point
mtc0            $26,$14             # write it in EPC register
```

C'est bien cette adresse située à l'adresse 0x01000000 qui est chargée dans le compteur ordinal.

Question E3

Si les adresses définies dans ces deux fichiers ne sont pas égales entre elles, le logiciel essaiera d'accéder à des adresses erronnées, puisque la base des segments pour le logiciel ne sera pas la même que la base des segments pour le matériel.

Question E4

Le segment seg_reset contient seulement le code de la fonction assembleur reset.

Le segment seg_kcode contient le code de la fonction assembleur giet (le point d'entrée du noyau) et le code de toutes les fonctions du noyau définies dans les fichiers drivers.c, common.c, ctx_handler.c, irq_handler.c, sys_handler.c et exc_handler.c. Lors de la compilation, le code objet est préfixé d'une espèce de tag, .text, qui fait que l'éditeur de liens sait quelles fonctions mettre dans quel segment.

Question E5

D'après le fichier sys.bin.txt, le segment seg_reset va des adresses [0xbfc00000 ; 0xbfc00023], soit 36 octets.

D'après le fichier sys.bin.txt, le segment seg_kcode effectivement occupé va des adresses [0x80000180 ; 0x8000227c] compris (on ne compte pas la section .MIPS.abiflags), soit :

8448 octets

Question E6

```c
#include "stdio.h"

__attribute__((constructor)) void main()
{
  char c;
  char s[] = "\n Hello World! \n";

  while (1) {
      tty_puts(s);
      tty_getc(&c);
  }
}
```

Question E7

Bien évidemment que la boucle non-déterministe (dans le sens où on ne saura pas quand on en sortira) est dans la fonction utilisateur tty_getc et non pas dans la fonction noyau _tty_read ! Aucun programmeur système sain d'esprit n'implémenterait une boucle non-déterministe dans du code noyau, surtout si le noyau est non-préemptif, multi-processus, en temps partagé (ce qui est le cas du GIET).

Les raisons pour ne pas faire ça sont légion, mais la plus évidente est la suivante : on a vu que les interruptions matérielles étaient masquées pendant l'exécution des fonctions noyau. Autrement dit, si la boucle avait été mise dans la fonction noyau, le processeur reste dans cette fonction noyau, oublieux de toutes les interruptions matérielles, même des interruptions horloge, **et rien ne peut plus l'en faire sortir** (sinon bien entendu le fait que l'utilisateur décide d'appuyer sur une touche de son clavier, ce qui peut très bien ne jamais arrivé). Ce processus qui exécute la fonction ne peut pas être tué, ne peut pas recevoir quelque signal que ce soit, et **ne peut pas être commuté**. Si il se trouve que le processeur est mono-coeur et mono-fil, alors la machine ne fait **rien d'autre** qu'attendre l'input de l'utilisateur.

En revanche, si on met la boucle dans le code utilisateur, le processus va faire plein d'appels système, mais il va régulièrement en sortir. Il peut donc recevoir des signaux, se faire tuer, se faire commuter si il prend trop de temps, etc... Autrement dit, le comportement qu'on attend d'un processus dans un noyau en multi-processus en temps partagé. Et surtout, les interruptions matérielles sont traitées par le système.

Question E8

D'après le fichier app.bin.txt, le segment seg_code occupe les adresses [0x400000 ; 0x40134f] comprises, soient :

4944 octets

Question E9

```
  .PHONY : all clean

  GIET_SYS_PATH=/users/enseig/alain/giet_2011/sys
  GIET_APP_PATH=/users/enseig/alain/giet_2011/app
  AS=/opt/gcc-cross-mipsel/8.2.0/bin/mipsel-unknown-elf-as
  CC=/opt/gcc-cross-mipsel/8.2.0/bin/mipsel-unknown-elf-gcc
  LD=/opt/gcc-cross-mipsel/8.2.0/bin/mipsel-unknown-elf-ld
  DU=/opt/gcc-cross-mipsel/8.2.0/bin/mipsel-unknown-elf-objdump
  APP_PATH=app
  SYS_PATH=sys

  all : sys.bin app.bin

  %.o : $(GIET_SYS_PATH)/%.c
    $(CC) -Wall -mno-gpopt -ffreestanding -mips32 -I$(GIET_SYS_PATH) -I. -c -o  $@ $<

  %.o : $(GIET_APP_PATH)/%.c
    $(CC) -Wall -mno-gpopt -ffreestanding -mips32 -I$(GIET_APP_PATH) -I. -c -o  $@ $<

  %.o : %.c
    $(CC) -Wall -mno-gpopt -ffreestanding -mips32 -I$(GIET_APP_PATH) -I. -c -o  $@ $<

  giet.o :
    $(AS) -g -mips32 -o giet.o $(GIET_SYS_PATH)/giet.s

  reset.o :
    $(AS) -g -mips32 -o reset.o reset.s

  sys.bin: reset.o giet.o drivers.o common.o ctx_handler.o irq_handler.o \
sys_handler.o exc_handler.o
    $(LD) -o sys.bin -T sys.ld reset.o giet.o drivers.o common.o \
ctx_handler.o irq_handler.o sys_handler.o exc_handler.o
    $(DU) -D sys.bin > sys.bin.txt

  app.bin : stdio.o main.o
    $(LD) -o app.bin -T app.ld stdio.o main.o
    $(DU) -D app.bin > app.bin.txt

  clean :
    rm -f *.o *.bin *.txt
```

Question F1

La première transaction sur le bus est une transaction rafale entre le maître, soit notre MIPS32 et la ROM, soit la target 0. C'est une requête en lecture des adresses 0xbfc00000, 0xbfc00004, 0xbfc00008, 0xbfc0000c qui correspondent bien aux adresses des quatre premières instructions de la fonction reset. La réponse à la première requête de la rafale est 0x27bd4000, soit l'instruction assembleur :

```
lui       sp,0x200
```

D'après ce qu'on comprend de la trace, la première instruction du code du boot est exécuté au cycle 10 (pour une numérotation commençant à 0, soit le 11ème cycle, et pour une latence de la RAM de 0) :

——- cycle = 10 ——-
bcu : fsm = IDLE
proc : <InsReq valid mode MODE_KERNEL @ 0xbfc00000>
proc : <InsRsp valid no error ins 0x3c1d0200>
proc : <DataReq invalid mode MODE_HYPER type DATA_READ @ 0 wdata 0 be 0>
proc : <DataRsp invalid no error rdata 0>
proc : ICACHE_IDLE DCACHE_IDLE PIBUS_IDLE
rom : IDLE
ram : IDLE
tty : IDLE keyboard status[0] = 0 display status[0] = 0
– pibus signals –
req = 0
gnt = 0
sel_rom = 0
sel_ram = 0
sel_tty = 0
avalid = 0
read = 0x1
lock = 0
address = 0xbfc0000c
ack = 0x2
data = 0x409a6000

C'est le premier cycle où InsRsp est valide, ce qui signifie qu'on a un cache hit sur l'adresse 0xbfc00000, et que donc le cache fournit la donnée 0x3c1d0200 au processeurs, qui la fait rentrer dans son pipeline.

La deuxième transaction sur le bus arrive au deuxième cache miss, soit au cycle 14 dans nos paramètres, parce qu'on a eu un cache miss sur l'adresse 0xbfc00010 :

——- cycle = 14 ————
bcu : fsm = IDLE
proc : <InsReq valid mode MODE_KERNEL @ 0xbfc00010>
proc : <InsRsp invalid no error ins 0x409a6000>

Cette deuxième transaction est donc une requête rafale en lecture à destination de la ROM, pour les adresses 0xbfc00010, 0xbfc00014, 0xbfc00018, 0xbfc0001c. La réponse à la première requête de la rafale est 0x3c1a0100, ce qui correspond à l'instruction assembleur :

```
lui       k0,0x100
```

Question F2

La première instruction à exécuter en mode utilisateur est demandée au cycle 50. C'est l'instruction située à l'adresse 0x004012dc, ce qui est bien la première instruction de main d'après app.bin.txt.

On a un cache miss, ce qui fait qu'on a une requête rafale en lecture, vers la ram, sur les adresses 0x004012d0, 0x004012d4, 0x004012d8, 0x004012dc (on demande trois adresses qui ne nous intéressent en fait pas, on ramène toute une ligne de cache).

L'instruction est effectivement exécutée au **cycle 60** :

——- cycle = 60 ————
bcu : fsm = IDLE
proc : <InsReq valid mode MODE_USER @ 0x4012dc>
proc : <InsRsp valid no error ins 0x27bdffd0>

Question F3

L'instruction de lecture du début de la chaîne de caractères "LF Hello World! LF" est l'instruction d'adresse 0x004012f0, celle-ci rentre dans le pipeline au cycle 91, le cache miss est détecté un cycle plus tard, au cycle 92, le bus est alloué au maître au cycle 93, et la demande rafale en lecture des adresses 0x01000070, 0x01000074, 0x01000078 (qui contient les 4 caractères 'LF', ' ', 'H' et 'e' qui nous intéressent) et 0x0100007c (qui contient les 4 caractères 'l', 'l', 'o', ' ') est faite au cycle **94**, on peut donc dater la première transaction à ce cycle-là.

Question F4

On recherche le cycle auquel on a la première écriture d'un caractère vers le tty.

La question est ambigüe. Il peut s'agir soit du moment où l'instruction d'écriture du caractère entre dans le pipeline, soit du moment où elle est effectivement exécutée, soit du moment où la transaction d'écriture est lancée sur le bus.

La première instruction processeur d'écriture vers l'adresse 0x90000000 intervient au cycle 1160 (elle est rentrée dans le pipeline au cycle 1159) :

```
——- cycle = 1160 ————
bcu : fsm = IDLE
proc : <InsReq valid mode MODE_KERNEL @ 0x800007f4>
proc : <InsRsp valid no error ins 0x8fc20014>
proc : <DataReq valid mode MODE_KERNEL type DATA_WRITE @ 0x90000000 wdata 0xa be 0xf>
proc : <DataRsp valid no error rdata 0>
proc : ICACHE_IDLE DCACHE_WRITE_REQ PIBUS_IDLE
rom : IDLE
ram : IDLE
tty : IDLE keyboard status[0] = 0 display status[0] = 0
– pibus signals –
req = 0
gnt = 0
sel_rom = 0
sel_ram = 0
sel_tty = 0
avalid = 0
read = 0x1
lock = 0
address = 0x800007fc
ack = 0x2
data = 0xafc20014
```

La première transaction d'écriture sur le bus vers l'adresse 0x90000000 (soit la partie display du TTY) intervient au cycle 1163 :

```
——- cycle = 1163 ————
bcu : fsm = AD | selected target = 2
proc : <InsReq valid mode MODE_KERNEL @ 0x800007fc>
proc : <InsRsp valid no error ins 0xafc20014>
proc : <DataReq invalid mode MODE_KERNEL type DATA_READ @ 0x2003f6c wdata 0 be 0xf>
proc : <DataRsp invalid no error rdata 0>
proc : ICACHE_IDLE DCACHE_IDLE PIBUS_WRITE_AD
rom : IDLE
ram : IDLE
tty : IDLE keyboard status[0] = 0 display status[0] = 0
– pibus signals –
req = 0
gnt = 0
sel_rom = 0
sel_ram = 0
```

sel_tty = 0x1
avalid = 0x1
read = 0
lock = 0
address = 0x90000000
ack = 0x2
data = 0xafc20014