M1 SESI 2017-2018
Architecture Multi-Processeurs
TP2 : Déploiement de code sur processeur programmable

Kevin Mambu

February 22, 2018

Énoncé du TP

Objectif : exécution d'une application logicielle en C sur une architecture comportant un processeur MIPS32.

- Cross-compilateur gcc-mipsel
- Chargement du fichier binaire

Propriétés de l'Architecture

- Processeur MIPS32 Pipeliné (5 étages)
- Cache d'instruction :
  - Direct Mapping
  - #ligne = 16 octets
  - #cache = 1 Ko
  - #lignes = 64 lignes
- Cache de données :
  - Mêmes spécifications que le cache d'instruction
  - Politique Write-Through
- Le contrôleur déclenche une transaction sur le PIBUS dnas les cas suivants :
  - miss instruction : demande d'instruction non-chargée

$$MISS\_INST_{\#cy} = \underbrace{1}_{detectMISS} + \underbrace{4*1}_{Rafale} + \underbrace{1}_{dernierACK}$$

  - miss data : $MISS\_DATA_{\#cy} = MISS\_INST_{\#cy}$
  - read uncached : lecture de donnée non-cachable (dépend du segment lu)

Le GIET

Le Gestionnaire d'Interruption, d'Exception et de Trappes (GIET) est le système d'exploitation utilisé sur cette architecture lors de ce TP.
S'il ne fournit pas le support de mémoire virtuelle et de création dynamique de tâches, il support les architectures multi-processeurs, permettant quand même l'exécution concurrente de tâches créées statiquement.

- sys_hander.c : gestionnaire d'appels systèmes
- exc_handler.c: gestionnaire d'exceptions
- irq_handler.c: gestionnaire d'interruptions
- ctx_handler.c: context switcher (mutiplexage temporel)
- drivers.c : méthodes d'accès aux périphériques

Boot Record, Bootloader et *Soclib::Loader*

Le Bootloader est un programme stocké en ROM, lancé au démarrage et chargé entre autres des tâches suivantes :

- vérification du bon fonctionnement du matériels
- tests sur le système
- populer la mémoire vive du système d'exploitation (chargement DD → RAM)

Parce que le rapport temporel entre la machine simulée et la machine simulante est élevée (1MHz contre 1GHz), SoCLib fournit un objet de programmation permettant de pré-charger par la machine simulante la mémoire vive avant simulation, pour faire abstraction du temps de chargement : il s'agit de la classe *SoCLib::Loader*.
Le loader peut charger différents types de fichiers, dont deux en particulier :

- les Binary Large OBject (blobs), des paquets binaires non-structurés.
- les ficheiers binaires générés par GCC (.bin, .elf) avec des métadonnées en entête.

Memory Map

segment	adresse	taille	composant
seg_tty	0x9000 0000	16 Ko	PIBUS_MULTI_TTY
seg_reset	0xBFC0 0000	4 Ko	ROM
seg_kcode	0x8000 0000	16 Ko	RAM
seg_kunc	0x8100 0000	4 Ko	RAM
seg_kdata	0x8200 0000	64 Ko	RAM
seg_code	0x0040 0000	16 Ko	RAM
seg_data	0x0100 0000	16 Ko	RAM
seg_stack	0x0200 0000	16 Ko	RAM

Répertoire soft/Dépendances

```
- main.c : code de l'application utilisateur
| main.o
  |+stdio.o
  -> app.bin

- reset.s: code de boot
| reset.o
  |+giet.o
  |+drivers.o
  |+common.o
  |+ctx_handler.o
  |+irq_handler.o
  |+sys_handler.o
  |+exc_handler.o
  -> sys.bin

nb : il a fallu ajouter au fichier config.h :
  #define NO_HARD_CC 0
pour mener a bien la compilation, cette option specifie si on veut desactiver
ou non la coherence des caches pour la simulation. Mais ici la valeur de cette
option n'importe pas, vu qu'il y a qu'un seul composant maitre qui converse
avec des cibles
```

filetree.txt

MIPS - Table des Registres

Name	Register Number	Usage	Preserved on call
$zero	0	the constant value 0	n.a.
$at	1	reserved for the assembler	n.a.
$v0-$v1	2-3	value for results and expressions	no
$a0-$a3	4-7	arguments (procedures/functions)	yes
$t0-$t7	8-15	temporaries	no
$s0-$s7	16-23	saved	yes
$t8-$t9	24-25	more temporaries	no
$k0-$k1	26-27	reserved for the operating system	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

Question C1

```
size_t  icache_ways      = 1;  // instruction cache number of ways
size_t  icache_sets      = 64; // instruction cache number of sets
size_t  icache_words     = 4;  // instruction cache number of words per line
size_t  dcache_ways      = 1;  // data cache number of ways
size_t  dcache_sets      = 64; // data cache number of sets
size_t  dcache_words     = 4;  // data cache number of words per line
size_t  wbuf_depth       = 8;  // write buffer depth
```

tp2_top.cpp

Question C2

seg_reset contient le code **boot** et doit être chargé de manière systématique à chaque démarrage de machine. De ce fait, il doit être stocké dans une mémoire statique et non-modifiable (Read-Only Memory), alors que les segments (*kcode, kdata*, etc) sont dans une mémoire volatile et potentiellement modifiable (Random Access Memory).

Question C3

Le segment *seg_tty* ne doit pas être cachable car les valeurs de ses registres évoluent indépendamment des composants maîtres qui veulent y accéder.

Question C4

Les segments protégés sont (*seg_reset, seg_kcode, seg_kunc, seg_kdata, seg_tty*).
Ces segments sont délimités des segments utilisateurs par leur adressage:

- L'espace utilisateur correspond à la moitié basse $[0x00000000, 0x7FFFFFFF]$ de l'espace d'adressage.
- L'espace privilégié correspond à la moitié haute $[0x80000000, 0xFFFFFFFF]$ de l'espace d'adressage.

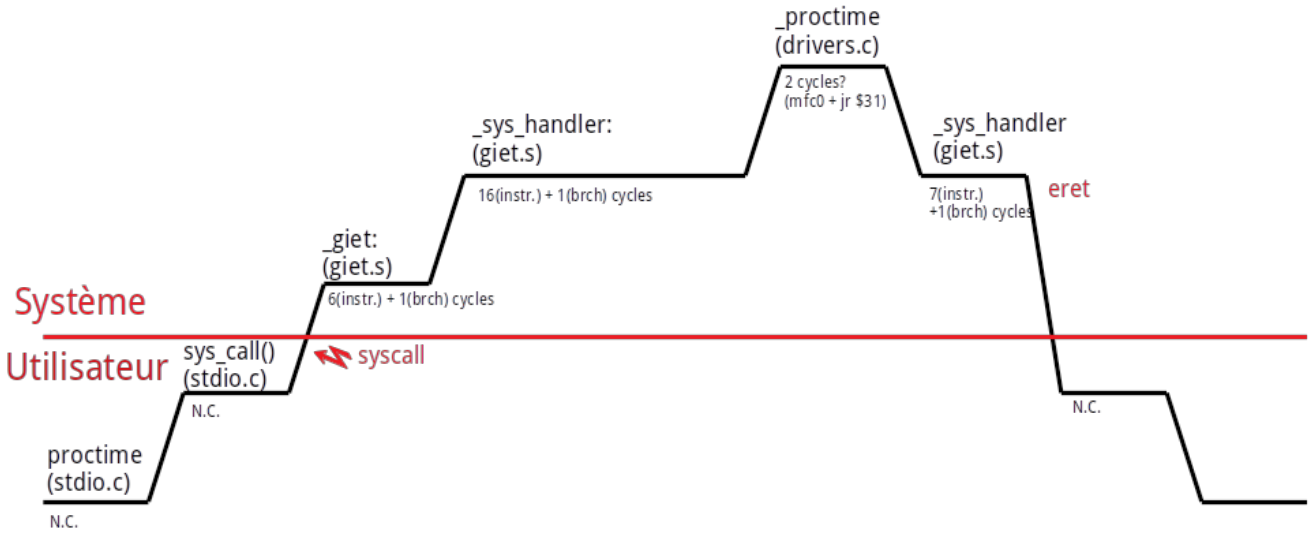
Question D1

Lors d'un appel système, un programme utilisateur doit fournir au systéme d'exploitation le numéro de l'appel système ainsi que ses arguments (4 max.).
Pour transmettre ces informations, la méthode *sys_call* charge ses paramétres en registres internes du processeur avant de faire un appel système.

Question D2

_cause_vector contient des valeurs, chaqu'une associée à un cas d'entrée au GIET possible. Il est initialisé dans le fichier *exc_handler.c*.
_syscall_vector contient les points d'entrées vers les adresses de chaque appel système. On pourrait dire qu'il s'agit d'une table de hash. Il est initialisé dans le fichier *sys_handler.c*.

Question D3/D4



Déterminer la durée d'exécution de *proctime()* est compliqué, vu que les temps en cycles de *sys_call()* et *proctime()* dépendent de la compilation.

Question E1

Le code de **boot** doit nécéssairement s'exécuter en mode superviseur pour trois raisons :

- Le segment qui lui est affilié par convention se trouve dans l'espace privilégié.
- Il doit charger en ménoire le système d'exploitation, or les segments affiliés par convention à l'OS se trouvent dans l'espace privilégié, le code de boot à donc besoin de privilèges particuliers pour pouvoir charger des données dans cette zone.

Question E2

Par convention le segment de code de l'application utilisateur, **code**, commence à l'adresse 0x0040 0000. C'est à cette adresse que le code de **boot** va systématiquement se brancher après avoir populé la RAM.

Question E3

Si les adresses de bases de segments spécifiées par le matériel sont différentes de celles spécifiées dans le logiciel utilisateur, ce dernier a de fortes chances de se faire tuer pour Erreur de Segmentation, en tentant d'acceder à une zone de l'espace d'adressage non-repertoriée.

Question E4

reset.o est chargé au segment **boot**. *giet.o* au segment **kcode**.

Question E5

seg_reset :

- première instr. à 0xBFC0 0000
- dernière instr. à 0xBFC0 0024
- 10 instructions, 40 octets

seg_kcode :

- première instr. à 0x8000 0000
- dernière instr. à 0x8000 2208
- 2179 instructions, 8716 octets

Question E6

```
#include "stdio.h"

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

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

} // end main
```

soft/main.c

Question E7

L'appel *tty_getc()* fonctionne comme suit : elle appele en boucle la fonction système *_tty_read()* et teste sa valeur de retour. Si cette dernière est différente de 0 alors la boucle est intérrompue.
Il faut mettre cette attente active dans *tty_getc()* et non *_tty_read()* parce que :

1. L'utilisateur fait appel à ce service donc ça doit être lui qui supporte les potentielles charges de temporisation, attendant la réponse à sa requête.
2. On ne peut pas se permettre un mécanisme de bloquage en mode système en lieu à cause de l'utilisateur. Ce dernier ne doit pas pouvoir entraver le bon fonctionnement de l'OS.

Question E8

seg_code :

- première instr. à 0x0040 0000
- dernière instr. à 0x0040 013C
- 1219 instructions, 4876 octets

Question E9

```
################################ COMPILATION TOOLS ################################
GIET_SYS_PATH=/users/enseig/alain/giet_2011/sys
GIET_APP_PATH=/users/enseig/alain/giet_2011/app
AS=/opt/gcc-cross-mipsel/4.3.3/bin/mipsel-unknown-elf-as
CC=/opt/gcc-cross-mipsel/4.3.3/bin/mipsel-unknown-elf-gcc
LD=/opt/gcc-cross-mipsel/4.3.3/bin/mipsel-unknown-elf-ld
DU=/opt/gcc-cross-mipsel/4.3.3/bin/mipsel-unknown-elf-objdump

################################ COMPILATION FLAGS ################################
APP_FLAGS=-Wall -mno-gpopt -ffreestanding -mips32 -I$(GIET_APP_PATH) -I. -c
SYS_FLAGS=-Wall -mno-gpopt -ffreestanding -mips32 -I$(GIET_SYS_PATH) -I. -c
AS_FLAGS=-g -mips32

#################################### FILES ########################################
GIET_SYS_OBJS   = drivers.o\
          common.o\
          ctx_handler.o\
          irq_handler.o\
          sys_handler.o\
          exc_handler.o
SYS_OBJS    = reset.o giet.o

GIET_APP_OBJS = stdio.o
APP_OBJS    = main.o

###################################################################################
all          : sys.bin app.bin

################################ WILDCARD RULES ###################################
$(GIET_SYS_OBJS)   : %.o : $(GIET_SYS_PATH)/%.c
  $(CC) $(SYS_FLAGS) -o  $@ $<

$(GIET_APP_OBJS)   : %.o : $(GIET_APP_PATH)/%.c
  $(CC) $(APP_FLAGS) -o  $@ $<

################################ SPECIFIC RULES ###################################
reset.o        : reset.s
  $(AS) $(AS_FLAGS) -o $@ $<

giet.o         : $(GIET_SYS_PATH)/giet.s
  $(AS) $(AS_FLAGS) -o $@ $<

main.o         : main.c
  $(CC) $(APP_FLAGS) -o  main.o main.c

############################## MAIN TARGET RULES ##################################
sys.bin        : $(SYS_OBJS) $(GIET_SYS_OBJS)
  $(LD) -o $@ -T sys.ld $^

app.bin        : $(GIET_APP_OBJS) $(APP_OBJS)
  $(LD) -o $@ -T app.ld $^

####################### META-RULES (clean,dump files) #############################
sys_dump       : sys.bin
  $(DU) -D sys.bin > sys.bin.txt

app_dump       : app.bin
  $(DU) -D app.bin > app.bin.txt

clean        :
  rm -rf app.bin sys.bin *.o *.bin.txt

.PHONY         : clean
```

soft/Makefile

Question F1

La première transaction sur le bus est à la suite d'un *miss instruction*, le miss est compulsif car le cache est encore vide à ce moment. Le processeur va demander les instructions situées en ROM.
Cela arrive dès le cycle 0 :

```
******* cycle = 0 **************
bcu : fsm = IDLE
proc : <InsReq     valid mode MODE_KERNEL @ 0xbfc00000>
proc : <InsRsp  invalid no error ins 0>
proc : <DataReq invalid mode MODE_HYPER type DATA_READ @ 0 wdata 0 be 0>
proc : <DataRsp invalid no error rdata 0>
proc : ICACHE_MISS_SELECT  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    = 0
lock    = 0
address = 0
ack     = 0
data    = 0
```

trace.txt

La première instruction du boot est :

```
bfc00000: 3c1d0200  lui sp,0x200
```

sys.bin.txt

Cela se fait au cycle 10 :

```
******* 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    = 0x341aff13
```

trace.txt

La deuxième transaction s'effectue au cycle 14 :

```
******* cycle = 14 **************
bcu : fsm = IDLE
proc : <InsReq     valid mode MODE_KERNEL @ 0xbfc00010>
proc : <InsRsp  invalid no error ins 0x341aff13>
proc : <DataReq invalid mode MODE_HYPER type DATA_READ @ 0 wdata 0 be 0>
proc : <DataRsp invalid no error rdata 0>
proc : ICACHE_MISS_SELECT  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    = 0x341aff13
```

trace.txt

Cela correspond à cette instruction (note : après le 3e cours on sait maintenant qu'il s'agit d'une ecriture externe) :

```
bfc00010: 409a6000   mtc0   k0,c0_status
```

sys.bin.txt

Question F2

La première instruction du main est :

```
004012dc <main>:
  4012dc: 27bdffd0   addiu sp,sp,-48
```

app.bin.txt

Cela se fait au cycle 57 :

```
*******  cycle = 57 **************
bcu : fsm = IDLE
proc : <InsReq     valid mode MODE_USER @ 0x400000>
proc : <InsRsp     valid no error ins 0x27bdffd0>
proc : <DataReq invalid mode MODE_KERNEL type DATA_READ @ 0x1000000 wdata 0 be 0xf>
proc : <DataRsp invalid no error rdata 0x400000>
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 = 0x40000c
ack     = 0x2
data    = 0x3a0f021
```

trace.txt

Question F3

La première transaction corresondant à la lecture de la chaine de caractères "Hello World!" se fait au cycle 32 :

```
*******  cycle = 32 **************
bcu : fsm = DTAD | selected target = 1
proc : <InsReq     valid mode MODE_KERNEL @ 0xbfc00020>
proc : <InsRsp   invalid no error ins 0x8f5a0000>
proc : <DataReq    valid mode MODE_KERNEL type DATA_READ @ 0x1000000 wdata 0 be 0xf>
proc : <DataRsp invalid no error rdata 0>
proc : ICACHE_MISS_WAIT   DCACHE_MISS_WAIT   PIBUS_READ_DTAD
rom : IDLE
ram : READ_OK
tty : IDLE    keyboard status[0] = 0    display status[0] = 0
  -- pibus signals --
req     = 0
gnt     = 0
sel_rom = 0
sel_ram = 0x1
sel_tty = 0
avalid  = 0x1
read    = 0x1
lock    = 0x1
address = 0x1000008
ack     = 0x2
data    = 0x6548200a
```

trace.txt

Question F4

L'appel de tty_puts() entraine un appel à la commande du pilote du TTY _tty_write() :

```
8000061c <_tty_write>:
```

sys.bin.txt

Dans _tty_write(), l'instruction d'ecriture d'un caractère est :

```
8000070c:  ac430000   sw   v1,0(v0)
```

sys.bin.txt

Je n'ai pas réussi à retrouver cette instruction dans la trace.