TP2 : Déploiement de code sur processeur programmable

Jiang 3602103
Li 3770906

C) Modélisation de l'architecture matérielle

Question C1 Quelles valeurs doivent prendre les paramètres icache_words, icache_sets, icache_ways, dcache_words, dcache_sets, dcache_ways , wbuf_depth pour donner aux caches les caractéristiques demandées ci-dessus?

	size	description
cache_words	4	## Headingnombre de mots par ligne
cache_sets	64	nombre de famille
cache_ways	1	nombre de cases par famille
wbuf_depth	8	size de 2 mots

Réponse : On choisira des lignes de cache de 4 mots de 32 bits, #words=4, pas d'associativité, #ways = 1, et une capacité totale de 1 Koctets pour chacun des deux caches, # sets=1024/16=64. On choisira une profondeur de 8 mots pour le tampon d'écritures postées.

Question C2 Pourquoi le segment seg_reset n'est-il pas assigné au même composant matériel que les 6 autres segments mémoire.

Réponse :
On démarre le système à base du programme dans seg_reset, donc il doit être accessible avant l'initialisation. On ne peut pas perdre les informations quand on éteint l'ordinateur, donc il faut l'assigner au ROM pour les garder.
Question C3 Expliquer pourquoi le segment seg_tty doit être non cachable.
Réponse : Non cachable : C'est-à-dire les données ne peuvent pas mettre dans la cache. Parce que les données de TTY doit changer en temp réel avec le périphérique.
On met la copie dans la cache et parfois ce ne pas mise à jour. Normalement on met les données de périphérique non cachable.

Question C4 Parmi les 8 segments utilisés dans cette architecture, quels sont les segments protégés (c'est à dire accessibles seulement quand le processeur est en mode superviseur). Comment est réalisée cette protection ?

Réponse : Les segments protégés : Kcode/ Kunc/Kdata/TTY, pour les protéger, ils sont contrôlé uniquement par le système d'exploitation.
Des adresses sont plus grandes que 0x8000000.

Complétez le fichier tp2_top.cpp pour définir la segmentation de de l'espace adressable. Il faut définir les valeurs des adresses de base et longueurs des segments au début du fichier, et il faut compléter la table des segments.

```
// segment definition
#define SEG_RESET_BASE  0xBFC00000 //address biger than 0x80000000 is protected
#define SEG_RESET_SIZE  0x00001000 //boot codes  ROM 4k

#define SEG_KCODE_BASE  0x80000000
```

```
#define SEG_KCODE_SIZE   0x00004000 //OS codes    RAM 16k


#define SEG_KDATA_BASE   0x82000000
#define SEG_KDATA_SIZE   0x00010000 //OS cachable codes       RAM 64k


#define SEG_KUNC_BASE    0x81000000
#define SEG_KUNC_SIZE    0x00001000 //OS uncachable codes RAM 4k


#define SEG_DATA_BASE    0x01000000
#define SEG_DATA_SIZE    0x00004000 //app globle data RAM 16k


#define SEG_CODE_BASE    0x00400000
#define SEG_CODE_SIZE    0x00004000 //app data        RAM 16k ( main() )


#define SEG_STACK_BASE   0x02000000
#define SEG_STACK_SIZE   0x00004000 //stack   RAM 16k
```

```
#define SEG_TTY_BASE    0x90000000
#define SEG_TTY_SIZE    0x00000010 //register of PIBUS_MULTI_TTY RAM 16  // uncachabe
```

```
    ////////////////////////////////////////////////
    //  SEGMENT TABLE DEFINITION
    ////////////////////////////////////////////////
    PibusSegmentTable   segtable;
    segtable.setMSBnumber(8);
    segtable.addSegment("seg_reset"  , SEG_RESET_BASE  , SEG_RESET_SIZE , ROM_INDEX,
true );
    segtable.addSegment("seg_kcode"  , SEG_KCODE_BASE  , SEG_KCODE_SIZE , RAM_INDEX,
true );
    segtable.addSegment("seg_kdata"  , SEG_KDATA_BASE  , SEG_KDATA_SIZE , RAM_INDEX,
true );
    segtable.addSegment("seg_kunc"   , SEG_KUNC_BASE   , SEG_KUNC_SIZE  , RAM_INDEX,
false);
    segtable.addSegment("seg_code"   , SEG_CODE_BASE   , SEG_CODE_SIZE  , RAM_INDEX,
true );
    segtable.addSegment("seg_data"   , SEG_DATA_BASE   , SEG_DATA_SIZE  , RAM_INDEX,
true );
    segtable.addSegment("seg_stack"  , SEG_STACK_BASE  , SEG_STACK_SIZE , RAM_INDEX,
true );
    segtable.addSegment("seg_tty"    , SEG_TTY_BASE    , SEG_TTY_SIZE   , TTY_INDEX,
false);
```

Complétez dans le fichier tp2_top.cpp le constructeur du composant loader qui permet de pré-charger dans les deux composants rom et ram le code binaire qui sera exécuté par le processeur MIP32.

```
    ////////////////////////////////////////////////////
    //  INSTANCIATED  COMPONENTS
    ////////////////////////////////////////////////////
    soclib::common::Loader  loader(sys_path, app_path);     //COMPLETED____________
    PibusSegBcu         bcu ("bcu", segtable, 1 , 3, 100);
    PibusMips32Xcache       proc    ("proc", segtable, 0,
                                     icache_ways, icache_sets, icache_words,
```

```
                                    dcache_ways, dcache_sets, dcache_words,
                                    wbuf_depth, snoop_active);
    PibusSimpleRam      rom ("rom"   , ROM_INDEX, segtable, 0, loader);
    PibusSimpleRam      ram ("ram"   , RAM_INDEX, segtable, ram_latency, loader);
    PibusMultiTty       tty     ("tty"   , TTY_INDEX, segtable, 1);
```

D) Système d'exploitation: GIET

Question D1 Quelles informations un programme utilisateur doit-il fournir au système d'exploitation lorsqu'il exécute un appel système ? Quelle est la technique utilisée pour transmettre ces informations?

Réponse : On met le numéro d'appel système dans le registre R2 et les 4 arguments dans les registres R4 à R7. Syscall adresse dans R3. (syscall handler dans giet.s)

Question D2 Ouvrez le fichier giet.s Que contiennent les deux tableaux _cause_vector[16] et _syscall_vector[32]? Par quoi sont-ils indexés? Dans quels fichiers ces tableaux sont-ils initialisés?

Réponse : Ils contiennent tous types de causes et tous types de syscall, ils sont indexés par numéro
_cause_vector[16]: les adresses des diffèrentes routines correspondant aux différentes causes des system call (indexé par champ XCODE dan cause register $13).(dans exc_handler.c)
_syscall_vector[32]:numéro d'appel de sys call (dans sys_handler.c)

Question D3 En analysant successivement le contenu des fichiers stdio.c, giet.s, sys_handler.c, drivers.c, donnez précisément la suite d'appels de fonctions déclenchée par l'appel système proctime().

Réponse :
proctime() dans studio.c ——>
sys_call() dans studio.c : executer assembleur syscall ——>
se brancher au GIET entry point 0x80000180 (dans giet.s) ——>
L'OS analyse le champ XCODE de syscall et appelle sys_handler (dans giet.s) ——>
Appelle _proctime()(vers _syscall_vector[32] */0x01/*) dans driver.c——>
executer mfc0 pour lire count.

Question D4 Donnez une estimation du coût (en nombre de cycles) de cet appel système, entre le branchement à la fonction proctime(), et le retour de la fonction.

Réponse : Dans stdio.c on consomme 1 cycle dans la fonction proctime() et 5 cycles dans sys_call() ; et puis 22 cycles dans la ficher giet.s ; 2 cycles dans drivers.c. Mais il y a le coût de miss, donc c'est pas certain.

E) Génération du code binaire

Question E1 Pourquoi le code boot doit-il nécessairement s'exécuter en mode superviseur?

Réponse : Parce que dans cette partie d'exécution du code dans zone protégé (kernel et périphérique), on veut obtenir tous les droits. Donc c'est en mode superviseur.
Complétez le fichier reset.s, pour initialiser le pointeur de pile (registre $29).

```
la           $29,    seg_stack_base
Addiu       $29,    $29,    0x00004000 # the base of stack is the biggest address
```

Question E2 Ce code de boot doit lancer l'application utilisateur. Quelle est la convention permettant au code de boot de récupérer l'adresse du point d'entrée dans le code applicatif (c'est à dire l'adresse de la première instruction de la fonction main() ?

Réponse : On écrit seg_data_base dans R14(EPC registre). _*attribute*_((constructor)) garanti que l'adresse de la fonction qui possède cet attribut doit être rangée au debut de seg_data.
Complétez le fichier seg.ld pour définir les adresses de base des 8 segments connus du logiciel.

```
seg_reset_base  = 0xBFC00000;/*TO BE COMPLETED*/

seg_kcode_base  = 0x80000000;
seg_kunc_base   = 0x81000000;
seg_kdata_base  = 0x82000000;



seg_code_base   = 0x00400000;
seg_data_base   = 0x01000000;
seg_stack_base  = 0x02000000;
```

```
seg_tty_base    = 0x90000000;
seg_timer_base  = 0x91000000;
seg_ioc_base    = 0x92000000;
seg_dma_base    = 0x93000000;
seg_gcd_base    = 0x95000000;
seg_fb_base     = 0x96000000;
seg_icu_base    = 0x9F000000;
```

Question E3 Que se passe-t-il si les adresses définies dans ces deux fichiers ne sont pas égales entre elles?

Réponse : Dans ce cas là le système d'exploitation ne peut pas accéder aux périphériques.

Question E4 En analysant le contenu du fichier sys.ld, déterminez quels sont les objets logiciels placés dans chacun des 2 segments qui contiennent du code système exécutable : seg_reset, seg_kcode.

Réponse : seg_reset : .reset -> par complication de reset.s
seg_kcode : .giet -> par complication de giet.s
.text -> par complication de *.c de l'OS

Question E5 En analysant le contenu du fichier sys.bin.txt, déterminez la longueur effective des deux segments seg_reset et seg_kcode.

Réponse :
seg_reset : 10 instructions = 40 octets
seg_kcode : base adresse : 0x8000180 , se termine à 0x80002214

Question E6 Complétez le fichier main.c

```c
#include "stdio.h"
__attribute__((constructor)) void main()
{
    char     c;
    char     s[] = "Hello World 20 chars";
    while(1)
        {
        tty_puts(s);
        tty_getc(&c);
    }
} // end main
```

Question E7 En analysant le code de l'appel système tty_getc() (que vous trouverez dans le ficher stdio.c) et le code de la fonction système _tty_read() (que vous trouverez dans le fichier drivers.c), expliquez le mécanisme qui rend cet appel système bloquant (c'est à dire qu'il ne rend la main au programme appelant que quand au moins un caractère a été saisi au clavier). En d'autre termes, laquelle des deux fonctions contient-elle la boucle d'attente? Expliquez pourquoi.

Réponse : tty_getc contient la boucle d'attente :
while (ret==0)…

Question E8 En analysant le contenu du fichier app.bin.txt, déterminez la longueur effective du segment seg_code.

Réponse : seg_code adresse : 0x400000-0x40134c

Question E9 Ecrivez un makefile qui automatise toutes les étapes de génération des deux fichiers sys.bin et app.bin.

```
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
CF =-Wall -mno-gpopt -ffreestanding -mips32 -I$(GIET_SYS_PATH) -I. -c -o
CFAPP =-Wall -mno-gpopt -ffreestanding -mips32 -I$(GIET_APP_PATH) -I. -c -o



.SUFFIXES:
.PHONY: clean
sys:
$(AS) -g -mips32 -o reset.o reset.s
$(AS) -g -mips32 -o giet.o $(GIET_SYS_PATH)/giet.s
$(CC)  $(CF)  drivers.o $(GIET_SYS_PATH)/drivers.c
$(CC)  $(CF)  common.o $(GIET_SYS_PATH)/common.c
$(CC)  $(CF)  ctx_handler.o $(GIET_SYS_PATH)/ctx_handler.c
$(CC)  $(CF)  irq_handler.o $(GIET_SYS_PATH)/irq_handler.c
$(CC)  $(CF)  sys_handler.o $(GIET_SYS_PATH)/sys_handler.c
$(CC)  $(CF)  exc_handler.o $(GIET_SYS_PATH)/exc_handler.c
$(LD) -o sys.bin -T sys.ld reset.o giet.o drivers.o common.o ctx_handler.o irq_handler.o sys_handler.o ex
$(DU) -D sys.bin > sys.bin.txt
app:
$(CC)  $(CFAPP)  stdio.o $(GIET_APP_PATH)/stdio.c
$(CC)  $(CFAPP) main.o main.c
$(LD) -o app.bin -T app.ld stdio.o main.o
$(DU) -D app.bin > app.bin.txt
clean:
rm -rf *.o
```

F) Exécution du code binaire sur le prototype virtuel

Question F1 En analysant la trace d'exécution, dites à quoi correspond la première transaction sur le bus ? A quel cycle le processeur exécute-t-il la première instruction du code de boot ? A quoi correspond la deuxième transaction sur le bus ?

Réponse : A cycle 7 le processeur exécute la première instruction du code de boot. Parce que à cycle 6 on peut voir que il y a read_ok dans ROM tandis que à cycle 7 l'état de ROM devient IDLE, c'est-à-dire on a déjà mis le code du root dans la cache pour l'exécuter. La première transaction sur le bus c'est pour la première partie du code de boot et la deuxième transaction est la deuxième partie du code de boot.

Question F2 A quel cycle s'exécute la première instruction de la fonction main() ?

Réponse :
Au cycle 48 req et gnt sont validés.
Et après au cycle 49 : sel_ram=0x1 et l'adresse correspond à laquelle de seg_code.
******* cycle = 48 **********

req = 0x1

gnt = 0x1

address = 0xbfc0002c

ack = 0x2

******* cycle = 49 **********

sel_ram = 0x1

address = 0x4012d0

Question F3 A quel cycle commence la première transaction correspondant à la lecture de la chaîne de caractères « hello world » ?

Réponse :

******* cycle = 51 **********

proc : PIBUS_READ_DTAD

Question F4 A quel cycle intervient la première écriture d'un caractère vers le terminal TTY ?

Réponse : A cycle 1209 on fait la première écriture d'un caractère vers TTY comme sel_tty =0x1 est l'adresse= 0x90000000.

******* cycle = 1209 **********

bcu : fsm = DT

proc : <InsReq valid mode MODE_KERNEL @ 0x8000071c>

proc : ICACHE_IDLE DCACHE_WRITE_UPDT PIBUS_WRITE_DT

tty : DISPLAY keyboard status[0] = 0 display status[0] = 0

address = 0x90000000

ack = 0x2

data = 0x48