**TP2 : Déploiement de code sur processeur programmable**

**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?**

**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.

|  |  |  |  |
| --- | --- | --- | --- |
| ICACHE | | DCACHE | |
| *icache\_words* | 4 | *dcache\_words* | 4 |
| *icache\_sets* | 64 | *dcache\_sets* | 64 |
| *icache\_ways* | 1 | *icache\_ways* | 1 |
| *wbuf\_depth* | | 8 | |

**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**

#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 exc\_handler.o

$(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