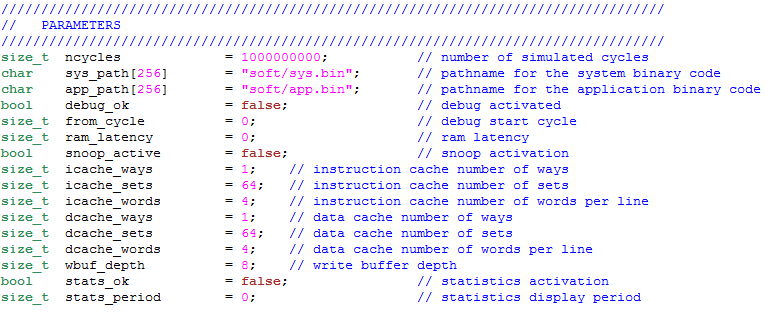
**TP2 : Déploiement de code sur processeur programmable**

**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 on ne peut pas perdre les informations quand on éteint l’ordinateur, donc il faut l’assigner au ROM pour les garder.

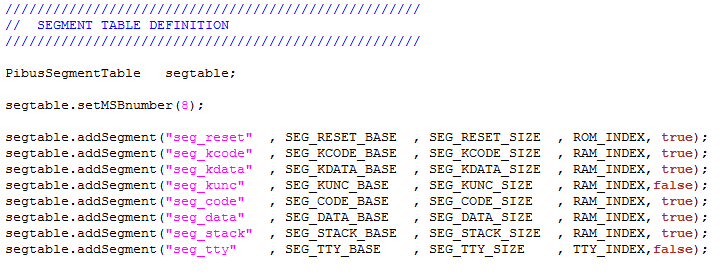
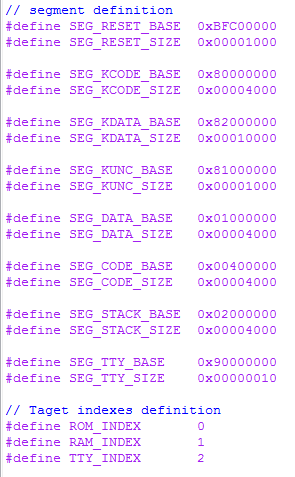
**Question C3** **Expliquer pourquoi le segment seg\_tty doit être non cachable.**

**Réponse :** Non cachable : C’est-à-dire les données ne peuvent pas mettre dans la câche. Parce qu’on met la copie dans la câche et parfois c’est faux. Normalement on met les données de périphérique non cachable.

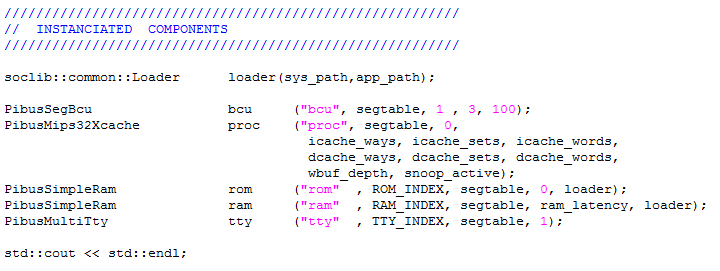
**Question C4** **Parmi les 8 segments utilisés dans cette architecture, quels sont les segments protégés (c'est à dire accessibles seulement quand le processeur est en mode superviseur). Comment est réalisée cette protection ?**

**Réponse :** Les segments protégés : Kcode/ Kunc/Kdata, pour les protéger, ils sont contrôlé uniquement par le système d’exploitation et ils ne peuvent pas être touché par l’utilisateur.

**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.**



**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.**



## 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.

**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, dans les fichiers exc\_handler et sys\_handler respectivement.

**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()->sys\_call()->\_proctime()

**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.**

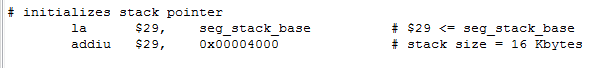
**Réponse :** Dans stdio.c on consomme 1 cycle dans la fonction proctime() et 5 cycles dans sys\_call() ; et puis 22 cycles dans la ficher giet.s ; 2 cycles dans drivers.c.

## 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,on veut obtenir tous les droits. Donc c’est en mode superviseur.

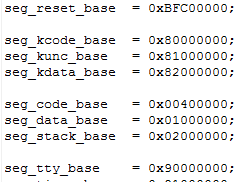
**Complétez le fichier reset.s, pour initialiser le pointeur de pile (registre $29).**



**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).

**Complétez le fichier seg.ld pour définir les adresses de base des 8 segments connus du logiciel.**



**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 ;

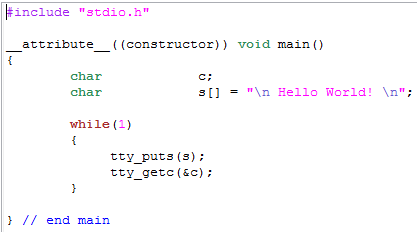
seg\_kcode : .giet & .text.

**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;

seg\_kcode : 2224.

**Question E6** **Complétez le fichier main.c**



**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 :** 134c.

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

## 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 :** A cycle 49 sel\_ram=0x1 et l’adresse correspond à laquelle de seg\_code.

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

**Réponse :** A cycle 30, sel\_ram=0x1, l’adresse=0x1000000 on a commencé à transmettre les données entre processeur et RAM pour faire la lecture de la chaîne de caractères.

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

**Réponse :** A cycle 1149 on fait la première écriture d’un caractère vers TTY comme sel\_tty =0x1 est l’adresse= 0x90000000.