

1. Récupération du code du TP
2. A. Travaux dirigés
  1. A1. Analyse de l'architecture
  2. A2. Programmation assembleur
  3. A2. Programmation en C
  4. A3. Compilation
3. B. Travaux pratiques
  1. B1. Premier programme en assembleur dans la seule section de boot
  2. B2. Saut dans le code du noyau en assembleur
  3. 3. Saut dans la fonction kinit() du noyau en langage C
  4. 4. Accès aux registres de contrôle des terminaux TTY
  5. 5. Premier petit pilote pour le terminal

## Boot et premier programme en mode kernel

Cette page décrit la séance complète : TD et TME. Elle commence par des exercices à faire sur papier et puis elle continue et se termine par des questions sur le code et quelques exercices de codage simples à écrire et tester sur le prototype. La partie pratique est découpée en 5 étapes. Pour chaque étape, nous donnons une brève description, suivie d'une liste des objectifs principaux et d'une liste des fichiers présents. Un bref commentaire est ajouté pour les fichiers. Vous avez une liste de questions simple et l'exercice de codage.

Avant de faire cette séance, vous devez avoir lu les documents suivants :

- Description des objectifs de cette séance et des suivantes : *obligatoire*
- Cours de démarrage présentant l'architecture matérielle et logicielle que vous allez manipuler *obligatoire*
- Document sur l'assembleur du MIPS et la convention d'appel des fonctions : *recommandé*
- Configuration de l'environnement des TP : *obligatoire*
- Documentation sur le mode kernel du MIPS32 : *optionnel pour cette séance*

## Récupération du code du TP

- Vous devez commencer par récupérer l'archive code du tp1
- Assurez-vous que vous avez déjà sourcé le fichier `Source-me.sh` (il doit être dans votre `.bashrc`).
- Placez cette archive dans le répertoire `AS5` et décompressez-la avec `tar xvzf tp1.tgz`
- Après décompression, avec la commande `tree -L 1 tp1`, vous devriez obtenir ceci:

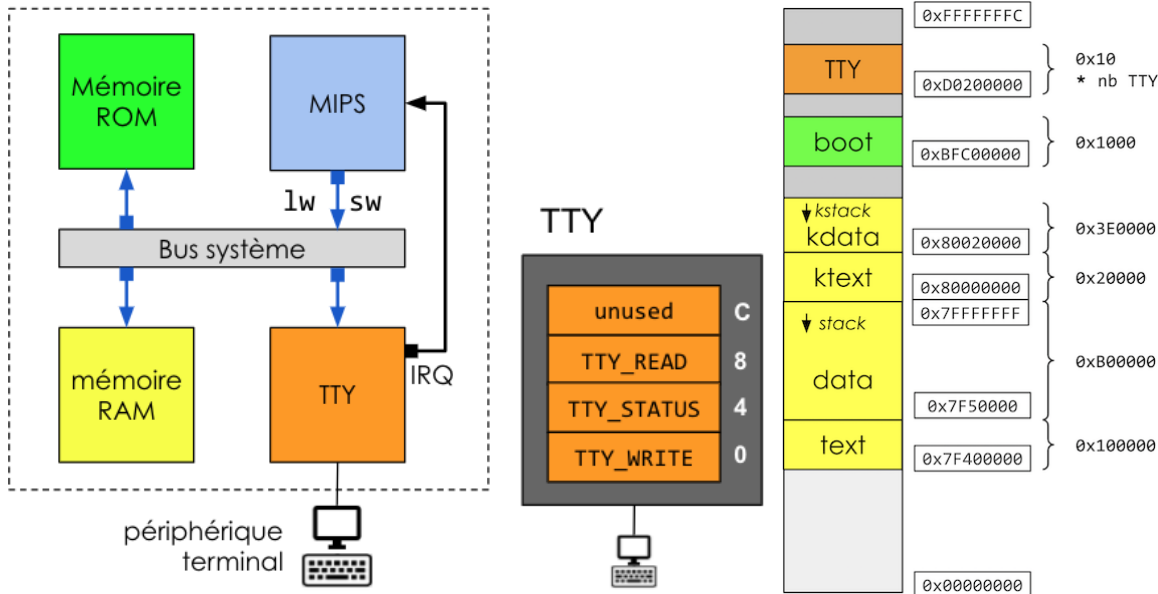
```
tp1
  ??? 1_hello_boot
  ??? 2_init_asm
  ??? 3_init_c
  ??? 4_nttys
  ??? 5_driver
  ??? Makefile
```

## A. Travaux dirigés

# A1. Analyse de l'architecture

Les trois figures ci-dessous donnent des informations sur l'architecture du prototype **almo1** sur lequel vous allez travailler.

- À droite, vous avez un schéma de connexion simplifié.
- Au centre, vous avez la représentation des 4 registres internes du contrôleur de terminal TTY nécessaires pour commander un couple écran-clavier.
- À gauche, vous avez la représentation de l'espace d'adressage implémenté pour le prototype.



## Questions

1. Il y a deux mémoires dans **almo1** : RAM et ROM. Qu'est-ce qui les distinguent et que contiennent-elles ?
2. Qu'est-ce l'espace d'adressage du MIPS ? Quelle taille fait-il ?  
Quelles sont les instructions du MIPS permettant d'utiliser ces adresses ? Est-ce synonyme de mémoire ?
3. Dans quel composant matériel se trouve le code de démarrage et à quel adresse est-il placé dans l'espace d'adressage et pourquoi à cette adresse ?
4. Quel composant permet de faire des entrées-sorties dans **almo1** ?  
Citez d'autres composants qui pourraient être présents dans un autre SoC ?
5. Il y a 4 registres dans le contrôleur de TTY, à quelles adresses sont-ils placés dans l'espace d'adressage ?  
Comme ce sont des registres, est-ce que le MIPS peut les utiliser comme opérandes pour ses instructions (comme `add`, `or`, etc.) ?  
Dans quel registre faut-il écrire pour envoyer un caractère sur l'écran du terminal (implicitement à la position du curseur) ?  
Que contiennent les registres `TTY_STATUS` et `TTY_READ` ?  
Quelle est l'adresse de `TTY_WRITE` dans l'espace d'adressage ?
6. Le contrôleur de TTY peut contrôler de 1 à 4 terminaux. Chaque terminal dispose d'un ensemble de 4 registres (on appelle ça une carte de registres, ou en anglais une register map). Ces ensembles de 4 registres sont placés à des adresses contiguës. S'il y a 2 terminaux (`TTY0` et `TTY1`), à quelle adresse est le registre `TTY_READ` de `TTY1` ?
7. Que représentent les flèches bleues sur le schéma ? Pourquoi ne vont-elles que dans une seule direction ?

## A2. Programmation assembleur

L'usage du code assembleur est réduit au minimum. Il est utilisé uniquement où c'est indispensable. C'est le cas du code de démarrage. Ce code ne peut pas être écrit en C au moins une raison importante. Le compilateur C suppose la présence d'une pile et d'un registre du processeur contenant le pointeur de pile, or au démarrage les registres sont vides (leur contenu n'est pas significatif). Dans cette partie, nous allons nous intéresser à quelques éléments de l'assembleur qui vous permettront de comprendre le code en TP.

### Questions

1. Nous savons que l'adresse du premier registre du TTY est `0xd0200000` est qu'à cette adresse se trouve le registre `TTY_WRITE` du `TTY0`. Écrivez le code permettant d'écrire le code ASCII 'x' sur le terminal 0. Vous avez droit à tous les registres du MIPS.
2. Le problème est que l'adresse du TTY est un choix de l'architecte du prototype et s'il décide de placer le TTY ailleurs dans l'espace d'adressage, il faudra réécrire le code précédent. Nous allons utiliser une étiquette, supposons que l'adresse du premier registre du TTY se nomme `__tty_regs_map`. Le code assembleur ne connaît pas l'adresse, il ne connaît que le symbole. Si nous voulons toujours écrire 'x' sur le terminal 0. Nous allons utiliser la macro `la $r, label` qui est remplacée par les deux instructions `lui` et `ori`. Il existe aussi la macro `li` pour initialiser des valeurs 32bits dans un registre. Pour être plus précis, les instructions

```
la $r, label
li $r, 0x87654321
```

sont remplacés par

```
lui $r, label>>16
ori $r, $r, label & 0xFFFF
lui $r, 0x8765
ori $r, $r, 0x4321
```

Réécrivez le code de la question précédente en utilisant `la` et `li`

3. En assembleur pour sauter à une adresse de manière inconditionnelle, on utilise les instructions `j label` ou `jr $r`, peuvent-elles faire les choses ?
4. Vous avez utilisé les directives `.text` et `.data` pour définir les sections où placer les instructions et les variables globales, mais il existe la possibilité de demander la création de nouvelle section dans le code objet produit par le compilateur avec la directive `.section name, "flags"`
  - ◆ `name` est le nom de la nouvelle section on met souvent un `.name` pour montrer que c'est une section et
  - ◆ `"flags"` informe du contenu `"ax"` pour des instructions, `"ad"` pour des données (ceux que ça intéresse pourront regarder là [?https://frama.link/20UzK0FP](https://frama.link/20UzK0FP))

Écrivez le code assembleur créant la section `".mytext"` et suivi de l'addition des registre `$5` et `$6` dans `$4`

5. A quoi sert la directive `.globl label` ?
6. Ecrivez une séquence de code qui affiche la chaîne de caractère `"Hello"` sur `TTY0`. Ce n'est pas une fonction et vous pouvez utiliser tous les registres que vous voulez. Vous supposez que `__tty_regs_maps` est déjà défini.
7. En regardant, le dessin de l'espace d'adressage du prototype **almo1**, à quelle adresse devra être initialisé le pointeur de pile pour le kernel. Rappelez pourquoi c'est indispensable de le définir avant d'appeler une fonction C et écrivez le code qui fait l'initialisation, en supposant que l'adresse du pointeur porte le nom `__kdata_end`

## A2. Programmation en C

Vous savez déjà programmer en C, mais vous allez voir des syntaxes ou des cas d'usage que vous ne connaissez peut-être pas encore. Les questions qui sont posées ici n'ont pas toutes été vues en cours, mais vous connaissez peut-être les réponses, sinon ce sera l'occasion d'apprendre.

### Questions

1. Quels sont les usages du mot clé `static` en C ?
2. Pourquoi déclarer des fonctions ou des variables `extern` ?
3. Comment déclarer un tableau de structures en variable globale ? La structure est nommée `test_s` a deux champs `int` nommés `a` et `b`. Le tableau est nommé `tab` a 2 cases.
4. Quelle est la différence entre `#include "file.h"` et `#include <file.h>` ?
5. Comment définir une macro instruction du processeur uniquement si elle n'est pas définie ? Écrivez un exemple.
6. Comment être certain de ne pas inclure plusieurs fois le même fichier `.h` ?
7. Supposons que la structure `tty_s` et le tableau de registres de TTY soient définis comme suit. Écrivez une fonction C `int getchar(void)` bloquante qui attend un caractère tapé au clavier sur le TTY0. Nous vous rappelons qu'il faut attendre que le registre `TTY_STATUS` soit différent de 0 avant de lire `TTY_READ`.

```
struct tty_s {
    int write;           // tty's output address
    int status;         // tty's status address something to read if not null)
    int read;           // tty's input address
    int unused;         // unused address
};
extern volatile struct tty_s __tty_regs_map[NTTYS];
```

8. Savez-vous à quoi sert le mot clé `volatile` ? Nous n'en avons pas parlé en cours, mais c'est nécessaire pour les adresses des registres de périphérique, une idée ... ?

## A3. Compilation

Pour obtenir le programme exécutable, nous allons utiliser :

- `gcc -o file.o -c file.c`
  - ◆ Appel du compilateur avec l'option `-c` qui demande à `gcc` de faire le préprocessing puis la compilation `c` pour produire le fichier objet `file.o`
- `ld -o bin.x -Tkernel.ld files.o ...`
  - ◆ Appel de l'éditeur de lien pour produire l'exécutable `bin.x` en assemblant tous les fichiers objets `.o`, en les plaçant dans l'espace d'adressage et résolvant les liens entre eux (quand un `.o` utilise une fonction ou une variable définie dans un autre `.o`).
- `objdump -D file.o > file.o.s` ou `objdump -D bin.x > bin.x.s`
  - ◆ Appel du désassembleur prend les fichiers binaires (`.o` ou `.x`) pour retrouver le code produit par le compilateur à des fins de debug ou de curiosité.

### Questions

Le fichier `kernel.ld` décrit l'espace d'adressage et la manière de remplir les sections dans le programme exécutable.

```

__tty_regs_map    = 0xd0200000 ;
__boot_origin    = 0xbfc00000 ;
__boot_length    = 0x00001000 ;
__ktext_origin   = 0x80000000 ;
__ktext_length   = 0x00020000 ;
[... question 1 ...]
__kdata_end      = __kdata_origin + __kdata_length ;

MEMORY {
    boot_region : ORIGIN = __boot_origin, LENGTH = __boot_length
    ktext_region : ORIGIN = __ktext_origin, LENGTH = __ktext_length
[... question 2 ...]
}

SECTIONS {
    .boot : {
        *(.boot)
    } > boot_region
[... question 3 ...]
    .kdata : {
        *(.*data*)
    } > kdata_region
}

```

1. Le fichier commence par la déclaration des variables donnant des informations sur les adresses et les tailles des régions de mémoire. Ces symboles n'ont pas de type et ils sont visibles de tous les programmes c, il faut juste leur donner un type pour le compilateur puisse les exploiter, c'est ce que nous avons fait pour `extern volatile struct tty_s __tty_regs_map[NTTYS]`. En regardant, dans le dessin de la représentation de l'espace d'adressage, complétez les lignes de déclaration des variables pour la région `kdata_region`
2. Le fichier contient ensuite la déclaration des régions qui vont être remplies par les sections trouvées dans les fichiers objets. Complétez les lignes propres à la déclaration de la région `kdata_region`.
3. Enfin le fichier contient comment sont remplies les régions avec les sections. Complétez les lignes correspondant à la description du remplissage de la région `ktext_region`. Vous devez la remplir avec les sections `.text` issus de tous les fichiers.

Nous allons systématiquement utiliser des Makefiles pour la compilation du code, mais aussi pour lancer le simulateur du prototype **almo1**. Pour cette première séance, les Makefiles ne permettent pas de faire des recompilations partielles de fichiers. Les Makefiles sont utilisés pour agréger toutes les actions que nous voulons faire sur les fichiers, c'est-à-dire : compiler, exécuter avec ou sans trace, nettoyer le répertoire. Nous avons recopiez le premier Makefile pour montrer sa forme et poser quelques questions, auxquels vous savez certainement répondre.

```

# Tools and parameters definitions
# -----
NTTY    ?= 2 #                               default number of ttys

CC      = mipsel-unknown-elf-gcc #          compiler
LD      = mipsel-unknown-elf-ld #          linker
OD      = mipsel-unknown-elf-objdump #     desassembler
SX      = almo1.x #                         prototype simulator

CFLAGS  = -c #                               stop after compilation, then produce .o
CFLAGS += -Wall -Werror #                   gives almost all C warnings and considers them to be errors
CFLAGS += -mips32r2 #                       define of MIPS version
CFLAGS += -std=c99 #                         define of syntax version of C
CFLAGS += -fno-common #                     do not use common sections for non-static vars (only bss)

```

```

CFLAGS += -fno-builtin #           do not use builtin functions of gcc (such as strlen)
CFLAGS += -fomit-frame-pointer #   only use of stack pointer ($29)
CFLAGS += -G0 #                   do not use global data pointer ($28)
CFLAGS += -O3 #                   full optimisation mode of compiler
CFLAGS += -I. #                   directories where include files like <file.h> are located
CFLAGS += -DNTTYS=$(NTTY) #       #define NTTYS with the number of ttys in the prototype

FROM   ?= 0 #                       first cycles to trace
NCYC   ?= 10000 #                   number of cycles to execute

# Rules (here they are used such as simple shell scripts)
# -----
help:
@echo "\nUsage : make <compil|exec|trace|clean> [FROM=num] [NCYC=num]\n"
@echo "      compil  : compile all sources"
@echo "      exec    : prototype execution"
@echo "      trace   : execution with a trace for debug (trace.log)"
@echo "              FROM is the first cycle from which the trace is made"
@echo "              NCYC is the number of cycles to execute"
@echo "      clean   : clean all compiled files\n"

compil:
$(CC) -o hcpu.o $(CFLAGS) hcpu.S
@$(OD) -D hcpu.o > hcpu.o.s
$(LD) -o kernel.x -T kernel.ld hcpu.o
@$(OD) -D kernel.x > kernel.x.s

exec: compil
$(SX) -KERNEL kernel.x -NTTYS $(NTTY)

trace: compil
$(SX) -KERNEL kernel.x -DEBUG $(FROM) -NCYCLES $(NCYC) -NTTYS $(NTTY) > details.log;\
echo "\ngenerate trace.log, please wait... ";\
tracelog -vout="trace.log.s" *.x.s details.log ;\
rm details.log

clean:
-rm *.o* *.x* *~ *.log.* proc?_term? 2> /dev/null || true

```

4. Au début du fichier se trouve la déclaration des variables du Makefile, quelle est la différence entre =, ?= et += ?
5. Ou est utilisé CFLAGS ? Que fait -DNTTYS=\$(NTTY) et pourquoi est-ce utile ici ?
6. Si on exécute make sans cible, que se passe-t-il ?
7. à quoi servent @ et - au début de certaines commandes ?

## B. Travaux pratiques

### B1. Premier programme en assembleur dans la seule section de boot

Nous commençons par un petit programme de quelques lignes en assembleur, placé entièrement dans la région mémoire du boot, qui réalise l'affichage du message "Hello World". C'est un tout petit programme, mais pour obtenir l'exécutable, vous devrez utiliser tous les outils de la chaîne de cross-compilation MIPS et pour l'exécuter vous devrez exécuter le simulateur du prototype. C'est simple, mais c'est nouveau pour beaucoup d'entre vous.

## Objectifs

- produire un exécutable à partir d'un code en assembleur.
- savoir comment afficher un caractère sur un terminal.
- analyse d'une trace d'exécution

## Fichiers

```
l_hello_boot
??? hcpu.S      : code dépendant du cpu matériel en assembleur
??? kernel.ld  : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
??? Makefile   : description des actions possibles sur le code : compilation, exécution, netto
```

## Questions

Les réponses sont dans le cours ou dans les fichiers sources. Certaines ont déjà été traitées plus haut.

1. Dans quel fichier se trouve la description de l'espace d'adressage du MIPS ? Que trouve-t-on dans ce fichier ?
2. Dans quel fichier se trouve le code de boot et pourquoi, selon vous, avoir nommé ce fichier ainsi ?
3. A quelle adresse démarre le MIPS ? Où peut-on le vérifier ?
4. Que produit `gcc` quand on utilise l'option `-c` ?
5. Que fait l'éditeur de liens ? Comment est-il invoqué ?
6. De quels fichiers a besoin l'éditeur de liens pour fonctionner ?
7. Dans quelle section se trouve le code de boot pour le compilateur ? (*la réponse est dans le code assembleur*)
8. Dans quelle section se trouve le message hello pour le compilateur ? Ce choix est particulier, mais ce message est en lecture seule.
9. Dans quelle section se trouve le code de boot dans le code exécutable ?
10. Dans quelle région de la mémoire le code de boot est-il placé ?
11. Comment connaît-on l'adresse du registre de sortie du contrôleur de terminal TTY ?
12. Le code de boot se contente d'afficher un message, comment sait-on que le message est fini et que le programme doit s'arrêter ?
13. Pourquoi terminer le programme par un `dead: j dead` ?

## Exercices

- Exécutez le programme en lançant le simulateur avec `make exec`, qu'observez-vous ?
- Exécutez le programme en lançant le simulateur avec `make trace`. Cela exécute le programme pour une courte durée et cela produit un fichier `trace.log.s`. Que voyez-vous dans la trace ?
- Modifiez le code de `hcpu.S` afin d'afficher le message "Au revoir\n" (*Hommage VGE*) après le message "Hello".  
Vous devez avoir deux messages, et pas seulement étendre le premier.

## B2. Saut dans le code du noyau en assembleur

Dans le deuxième programme, nous restons en assembleur, mais nous avons deux fichiers source : (1) le fichier contenant le code de boot et (2) le fichier contenant le code du noyau. Ici, le code du noyau c'est juste une *fonction* `kinit()`. Ce n'est pas vraiment une fonction car on n'utilise pas la pile. **Objectifs**

- Savoir comment le programme de boot fait pour sauter à l'adresse de la routine kinit.
- Avoir un fichier kernel.ld un peu plus complet.

## Fichiers

```

2_init_asm/
??? hcpu.S      : code dépendant du cpu matériel en assembleur
??? kernel.ld  : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
??? kinit.S    : fichier contenant le code de démarrage du noyau, ici c'est une routine kinit.
??? Makefile   : description des actions possibles sur le code : compilation, exécution, netto

```

## Questions

1. Regarder dans le fichier hcpu.S, dans quelle section est désormais le code de boot ?

- Le code de boot ne fait que sauter dans la fonction kinit avec l'instruction j,

il n'y a pas de retour, ce n'est donc pas un jal, mais pourquoi ne pas avoir utilisé j init et donc pourquoi passer par un registre ?

- Dans kernel.ld, la définition de la mémoire est plus complète, elle contient 3 régions : pour le code de boot boot\_region pour le code du noyau ktext\_region et pour les données globales du noyau kdata\_region. Ces régions ne contiennent qu'une section de sorties (resp. .boot, .ktext et .kdata) remplies avec les sections d'entrées produites par le compilateur.

Que signifie (\*.data\*) ?

- Quelle est la valeur de \_\_kdata\_end ? Pourquoi, selon vous, mettre 2 «\_» au début des variables ?

## 3. Saut dans la fonction kinit() du noyau en langage C

Dans ce troisième programme, nous faisons la même chose que pour le deuxième mais kinit() est désormais écrit en langage C. Cela change peu de choses, sauf une chose importante kinit() est une fonction et donc il faut absolument une pile d'exécution.

### Objectifs

- Savoir comment et où déclarer la pile d'exécution du noyau.
- Savoir comment afficher un caractère sur un terminal depuis un programme C.

### Fichiers

```

3_init_c/
??? hcpu.S      : code dépendant du cpu matériel en assembleur
??? kernel.ld  : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
??? kinit.c    : fichier en C contenant le code de démarrage du noyau, ici c'est la fon
??? Makefile   : description des actions possibles sur le code : compilation, exécution

```

### • Questions

*Les réponses sont dans le cours ou dans les fichiers sources*

- ◆ Question ?



## 4. Accès aux registres de contrôle des terminaux TTY

Le prototype de SoC que nous utilisons pour les TP est configurable. Il est possible par exemple de choisir le nombre terminaux texte (TTY). Par défaut, il y en a un mais, nous pouvons en avoir jusqu'à 4. Nous allons modifier le code du noyau pour s'adapter à cette variabilité. En outre, pour le moment, nous ne faisons qu'écrire sur le terminal, maintenant, nous allons aussi lire le clavier.

### Objectifs

- Savoir comment compiler un programme C avec du code conditionnel.
- Savoir comment décrire en C l'ensemble des registres d'un contrôleur de périphérique et y accéder.

### Fichiers

```
4_nttys/
??? hcpu.S      : code dépendant du cpu matériel en assembleur
??? kernel.ld   : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
??? kinit.c     : fichier en C contenant le code de démarrage du noyau, ici c'est la fon
??? Makefile    : description des actions possibles sur le code : compilation, exécution
```

### • Questions

*Les réponses sont dans le cours ou dans les fichiers sources*

◆ Question ?

## 5. Premier petit pilote pour le terminal

Dans l'étape 4, nous accédons au registre de périphérique directement dans la fonction `kinit()`, ce n'est pas très simple. C'est pourquoi nous allons ajouter un niveau d'abstraction qui représente un début de pilote de périphérique (device driver). Ce pilote, même tout petit constitue une couche logicielle avec une API.

### Objectifs

- Savoir comment créer un début de pilote pour le terminal TTY.
- Savoir comment décrire une API en C

### Fichiers

```
5_driver/
??? harch.c     : code dépendant de l'architecture du SoC, pour le moment c'est juste le
??? harch.h     : API du code dépendant de l'architecture
??? hcpu.S      : code dépendant du cpu matériel en assembleur
??? kernel.ld   : ldscript décrivant l'espace d'adressage pour l'éditeur de lien
??? kinit.c     : fichier en C contenant le code de démarrage du noyau, ici c'est la fon
??? Makefile    : description des actions possibles sur le code : compilation, exécution
```

### • Questions

*Les réponses sont dans le cours ou dans les fichiers sources*

◆ Question ?

- Quel est le nom de la directive assembleur permettant de déclarer une section