1. Récupération du code du TP
2. A. Travaux dirigés
    1. A1. Analyse de l'architecture
    2. A2. Programmation assembleur
    3. A3. Chaîne de compilation
    4. A4. Programmation en C
3. Travaux Pratiques
    1. 1. Premier programme en assembleur dans la seule section de boot=
    2. 2. 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 découpé 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:

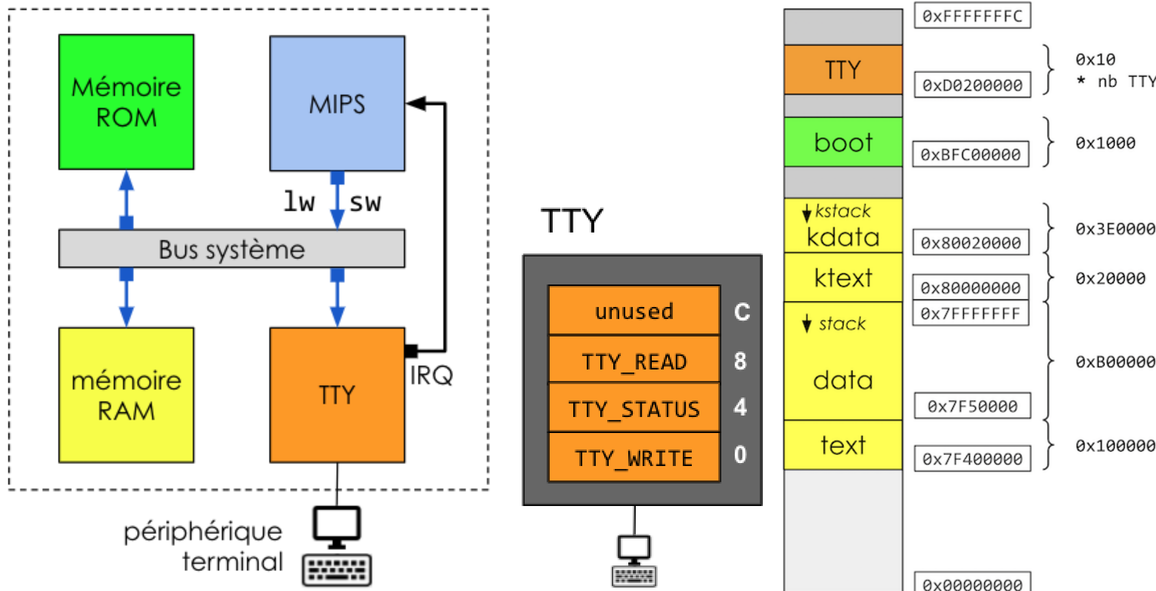
```
tp1
??? 1_hello_boot
??? 2_init_asm
??? 3_init_c
??? 4_nttys
??? 5_driver
??? Makefile
```

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 ? Quel 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`, a quelles adresses sont-il 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`), A 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 connait 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 la section où placer les instructions et les variables globales, mais il existe la possibilité de demander la création de directives 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
     ?https://frama.link/20UzK0FP
     ```
     lui $r, label>>16
     ori $r, $r, label & 0xFFFF
     lui $r, 0x8765
     ori $r, $r, 0x4321
     ```

A3. Chaîne de compilation

A4. Programmation en C

- fonction write buffer en assembleur et en C

- Makefile et make recurs
- Les struct et les tableaux de structs
- fonction read buffer en C
- les usages de extern/globl et de volatile
- compilation conditionnelle

Travaux Pratiques

1. Premier programme en assembleur dans la seule section de boot=

ajouter des analyses des fichiers objets et des traces

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 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**

```
1_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
```

- **Questions**
  Les réponse sont dans le cours ou dans les fichiers sources
    - Dans quel fichier se trouve la description de l'espace d'adressage du MIPS ? Que trouve-t-on dans ce fichier ?
    - Dans quel fichier se trouve le code de boot et pourquoi avoir nommé ce fichier ainsi ?
    - A quelle adresse démarre le MIPS ? Où peut-on le vérifier ?
    - Que produit `gcc` quand on utilise l'option `-c` ?
    - Que fait l'éditeur de liens ? Comment est-il invoqué ?
    - De quels fichiers a besoin l'éditeur de liens pour fonctionner ?
    - Dans quelle section se trouve le code de boot pour le compilateur ? *(la réponse est dans le code assembleur)*
    - Dans quelle section se trouve le message hello pour le compilateur ?
    - Dans quelle section se trouve le code de boot dans le code exécutable ?
    - Dans quelle région de la mémoire le code de boot est-il placé ?
    - Comment connaît-on l'adresse du registre de sortie du contrôleur de terminal `TTY` ?

- 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 ?
- Pourquoi terminer le programme par un `dead: j dead` ?

- **Exercice**
  - 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.

2. 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
??? Makefile      : description des actions possibles sur le code : compilation, exécution
```

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

  - 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 faisions 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