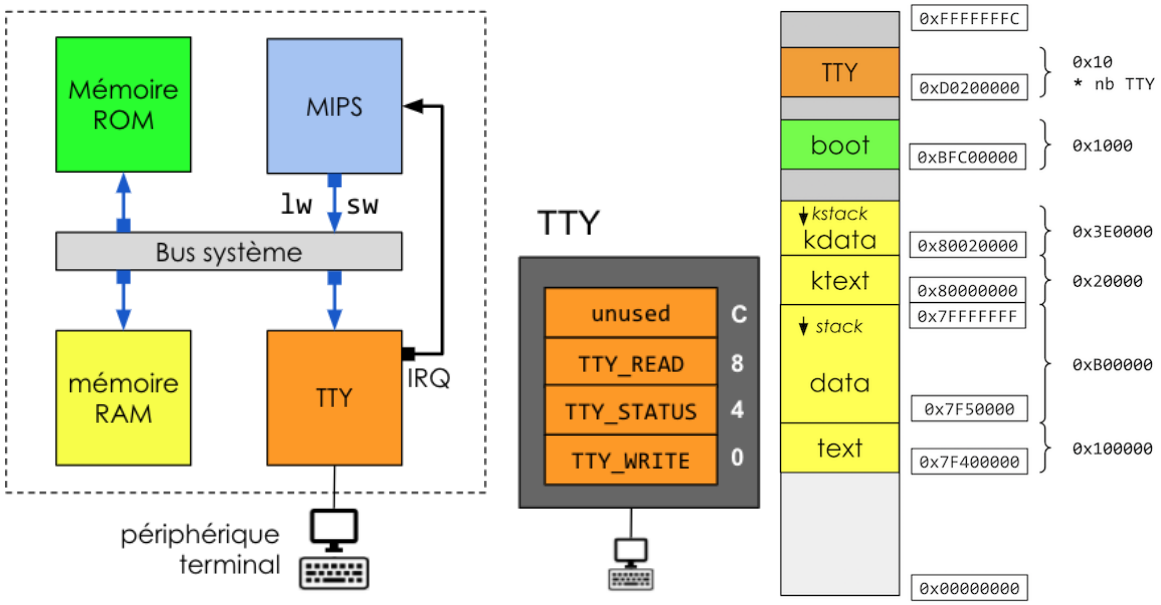
Boot et premier programme en mode kernel

1. Analyse de l'architecture

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

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



Questions

1. Il y a deux mémoires dans **almo1** : RAM et ROM. Qu'est-ce qui les distinguent et que contiennent-elles ? (C9 S6+S9)
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 ? (C9 S7)
3. Dans quel composant matériel se trouve le code de démarrage et à quelle adresse est-il placé dans l'espace d'adressage et pourquoi à cette adresse ? (C9 S6+S7)

4. Quel composant permet de faire des entrées-sorties dans almo1 ?
   Citez d'autres composants qui pourraient être présents dans un autre SoC ? (C9 S6+connaissances personnelles)
5. Il y a 4 registres de commande 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 ? (C9 S10)
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` ? (C9 S10)
7. Que représentent les flèches bleues sur le schéma ci-dessus ? Pourquoi ne vont-elles que dans une seule direction ? (C9 S11)

2. 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 pour au moins la raison suivante : le compilateur C suppose la présence d'une pile et d'un registre du processeur contenant le pointeur de pile, or au démarrage le contenu des registres 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`. (C9 S10)
   Écrivez le code permettant d'écrire le code ASCII `'x'` sur le terminal 0. Vous avez droit à tous les registres du MIPS puisqu'à ce stade il n'y pas de conventions sur leur utilisation.
   Ce qu'il faut bien comprendre, c'est que l'adresse du registre TTY_WRITE est l'adresse d'une sortie du SoC, ce n'est pas une mémoire à proprement parler. Il est d'ailleurs interdit de lire à cette adresse. Pour écrire un message à l'écran, il faut écrire tous les caractères du message à cette adresse (0xD0200000).
2. Dans la question précédente, on connaissait l'adresse absolue du registre TTY_WRTIE. On suppose désormais que l'adresse du premier registre du `TTY` se nomme `__tty_regs_map` (c'est le nom du label). Le programmeur ne connait pas l'adresse, il ne connaît que le nom du label (on dit aussi symbole). Ainsi, pour écrire `'x'` sur le terminal 0, nous devons utiliser la macro instruction `la $r, label`. Cette macro-instruction est remplacée lors de l'assemblage du code par les instructions `lui` et `ori`. Il existe aussi la macro instruction `li` qui demande de charger une valeur sur 32 bits dans un registre.

   Pour être plus précis, les macro-instructions
   ```
   la $r, label
   li $r, 0x87654321
   ```

   sont remplacées par

   ```
   lui $r, label>>16             // chargement des 16 bits de poids forts de label
   ori $r, $r, label & 0xFFFF    // chargement des 16 bits de poids faible de label
   ```

```
lui $r, 0x8765                  // chargement des 16 bits de poids fort de 0x8765431
ori $r, $r, 0x4321              // chargement des 16 bits de poids faible de 0x8765431
```

Réécrivez le code de la question précédente en utilisant `la` et `li` (C9 S34)

3. En assembleur pour sauter à une adresse de manière inconditionnelle, on utilise les instructions `j label` et `jr $r`. Ces instructions permettent-elles d'effectuer un saut à n'importe quelle adresse ?
   Attention, la réponse n'est pas dans le cours, mais dans la connaissance du codage des instructions de saut (`j`ump et `b`ranch) que vous avez vu au début du module.
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 d'une 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` (avec un `.` au début) pour montrer que c'est une section et
   - `"flags"` informe sur le contenu : `"ax"` pour des instructions, `"ad"` pour des données (ceux que ça intéresse pourront regarder le manuel de l'assembleur ?Assembleur/Directives/.section)

   Écrivez le code assembleur créant la section `".mytext"` et suivi de l'addition des registres `$5` et `$6` dans `$4` (C9 S10)
5. À quoi sert la directive `.globl label` dans un programme assembleur ? (C9 S34)
6. Écrivez une séquence de code qui affiche la chaîne de caractère `"Hello"` sur `TTY0`.
   Vous devez déclarer la chaine de caractère `"Hello"` dans la section `.data`, puis écrire le code dans la section `.text`. Vous pouvez utiliser tous les registres que vous voulez. Vous supposez que le label `__tty_regs_maps` est déjà défini et qu'il désigne le premier registre de commande du `TTY0` qui est `TTY_WRITE`.
   C'est une boucle qui écrit chaque caractère de la chaîne dans le registre TTY_WRITE du TTY0, jusqu'à trouver le `0` de fin de chaîne.
7. En regardant le dessin de l'espace d'adressage du prototype **almo1** (plus haut et sur le slide 7 du cours 9), dites 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 de pile a pour nom `__kdata_end`.

3. Compilation et édition de liens

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 liens 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.
    Autrement dit, quand un fichier `f1.o` utilise une fonction `funf2()` ou une variable `varf2` définie dans un autre fichier `f2.o`, alors l'éditeur de liens place dans l'espace d'adressage les sections `.text` et `.data` des fichiers `f1.o` et `f2.o`, puis il détermine alors quelles sont les adresses de `funf2()` et `varf2` dans l'espace d'adressage et il complètent les instructions de `f1` qui utilisent ces adresses.
- `objdump -D file.o > file.o.s` ou `objdump -D bin.x > bin.x.s`
  - Appel du désassembleur qui prend les fichiers binaires (`.o` ou `.x`) pour retrouver le code produit par le compilateur pour le debug.

Questions sur l'édition de lien

Le fichier `kernel.ld` décrit l'espace d'adressage et la manière de remplir les sections dans le programme exécutable. Ce fichier est utilisé par l'éditeur de lien. C'est un `ldscript`, c'est-à-dire un ?`script` pour `ld`.

```
__tty_regs_map   = [... question 1 ...] ;
__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 `kernel.ld` 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. 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`. Pour répondre, il faut savoir interpréter le dessin représentant l'espace d'adressage. (C9 S7+S38+S39)
2. Le fichier contient ensuite la déclaration des régions (dans `MEMORY{...}`), c'est-à-dire les segments d'adresse en mémoire qui seront remplies par l'éditeur de lien avec les sections trouvées dans les fichiers objets selon un ordre décrit dans la partie `SECTIONS{}` du `ldscript`. Complétez cette partie (la zone `[... question 2 ...]`) pour ajouter les lignes correspondant à la déclaration de la région `kdata_region` ? (C9 S38+S39)
3. Enfin le fichier décrit 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.
   Il faut bien comprendre que `.ktext` est une section produite par l'éditeur de liens. C'est ce que l'on appelle une section de sortie. `.text` est une section que l'éditeur de liens trouve dans un fichier objet `.o`, c'est ce que l'on appelle une section d'entrée. Comme il y a plusieurs fichiers objet, on doit dire à l'éditeur de lien de prendre toutes les sections `.text` de tous les fichiers qu'on lui donne. (C9 S38+S39)

3. Programmation en C

Vous savez déjà programmer en C, mais vous allez voir ici 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 ? (c'est une directive que l'on donne au compilateur 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`, elle a deux champs `int` nommés `a` et `b`. Le tableau est nommé `tab` et a 2 cases.
4. Supposons que la structure `tty_s` et le tableau de registres de `TTY` soient définis comme suit. Écrivez la fonction C `int getchar0(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`. `NTTYS` est un `#define` définit dans le Makefile de compilation avec le nombre de terminaux du SoC (en utilisant l'option `-D` de gcc). (C9 S10)

```
struct tty_s {
    int write;          // tty's output
    int status;         // tty's status something to read if not null)
    int read;           // tty's input
    int unused;         // unused
};
extern volatile struct tty_s __tty_regs_map[NTTYS];
// extern   : parce que ce tableau n'est pas dans ce fichier
// volatile : parce que le contenu du tableau peut changer tout seul, gcc doit le lire à
//            cela implique que gcc ne peut pas faire d'optimisation avec les registres d
//            (cf. note en fin de page)
```

5. Écrivez la fonction C `int puts0(char *s)` qui écrit tous les caractères de la chaîne `s` sur le terminal TTY0. La fonction doit rendre le nombre de caractères écrits. On suppose que les registres des TTYs sont définis comme dans la question précédente.

4. Usage de Make (optionnel)

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 recopié partiellement le premier Makefile pour montrer sa forme et poser quelques questions, auxquels vous savez certainement répondre.

La syntaxe des `Makefiles` peut-être très complexe (c'est un vieux langage), ici nous ne verrons qu'une petite partie. Notez que le Makefile `voit` les variables du `shell` comme s'il les avait définies lui-même.

```
# Tools and parameters definitions
# ------------------------------------------------------------------------------
# -- options used by the prototype simulator
NTTY   ?= 2 #                          default number of ttys

# -- tools
CC      = mipsel-unknown-elf-gcc #     cross-compiler MIPS
LD      = mipsel-unknown-elf-ld #      linker MIPS
OD      = mipsel-unknown-elf-objdump # desassembler MIPS
SX      = almo1.x #                    prototype simulator (nammed almo1.x)

# -- All flags used for the gcc compiler
CFLAGS  = -c #                         stop after compilation, then produce .o
CFLAGS += -Wall -Werror #              near all C warnings that becoming errors
CFLAGS += -mips32r2 #                  define of MIPS version
CFLAGS += -std=c99 #                   define of syntax version of C
```

```
CFLAGS += -fno-common #               no use common sections for nostatic vars
CFLAGS += -fno-builtin #              no use builtin func of gcc (ie strlen)
CFLAGS += -fomit-frame-pointer #      only use of stack pointer ($29)
CFLAGS += -G0 #                       do not use global data pointer ($28)
CFLAGS += -O3 #                       full optimisation mode of compiler
CFLAGS += -I. #                       dir. where include <file.h> are
CFLAGS += -DNTTYS=$(NTTY) #           number of ttys in the prototype

# Rules (here they are used such as simple shell scripts)
# ------------------------------------------------------------------------------
help:
    @echo "\nUsage : make <compil|exec|clean> [NTTY=num]\n"
    @echo "        compil  : compiles all sources"
    @echo "        exec    : executes the prototype"
    @echo "        clean   : clean all compiled files\n"

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

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

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

4. Où est utilisé `CFLAGS` ? Que fait `-DNTTYS=$(NTTY)` et pourquoi est-ce utile ici ? (C9 annexe S8)
5. Si on exécute `make` sans cible, que se passe-t-il ? (C9 annexe S6)

Réponses non présentes dans les slides, mais utiles à savoir.

6. à quoi servent `@` et `-` au début de certaines commandes ?
7. Au début du fichier se trouve la déclaration des variables du Makefile, quelle est la différence entre `=`, `?=` et `+=` ?

Note sur le mot clé `volatile`

- *Quand le programme doit aller chercher une donnée dans la mémoire puis faire plusieurs calculs dessus, le compilateur optimise en réservant un registre du processeur pour cette variable afin de ne pas être obligé d'aller lire la mémoire à chaque fois. Mais, il y a des cas où ce comportement n'est pas souhaitable (il est même interdit). C'est le cas pour les données qui se trouvent dans les registres de contrôleur de périphériques. Ces données peuvent être changées par le périphérique sans que le processeur le sache, de sorte qu'une valeur lue par le processeur à l'instant `t` n'est plus la même (dans le registre du périphérique) à l'instant `t+1`. Le compilateur ne doit pas optimiser, il doit aller chercher la donnée en mémoire à chaque fois que le programme le demande.*

    - `volatile` *permet de dire à `gcc` que la variable en mémoire peut changer à tout moment, elle est volatile. Ainsi quand le programme demande de lire une variable `volatile` le compilateur doit toujours aller la lire en mémoire. Il ne doit jamais chercher à optimiser en utilisant un registre afin de réduire le nombre de lecture mémoire (load). De même, quand le programme écrit dans une variable `volatile`, cela doit toujours provoquer une écriture dans la mémoire (store).*

◊ *Ainsi, les registres de périphériques doivent toujours être impérativement lus ou écrits à chaque fois que le programme le demande, parce que c'est justement ces lectures et ces écritures qui commandent le périphérique.*