Prise en main du système
Hello World

MI074 - 2

Objectif

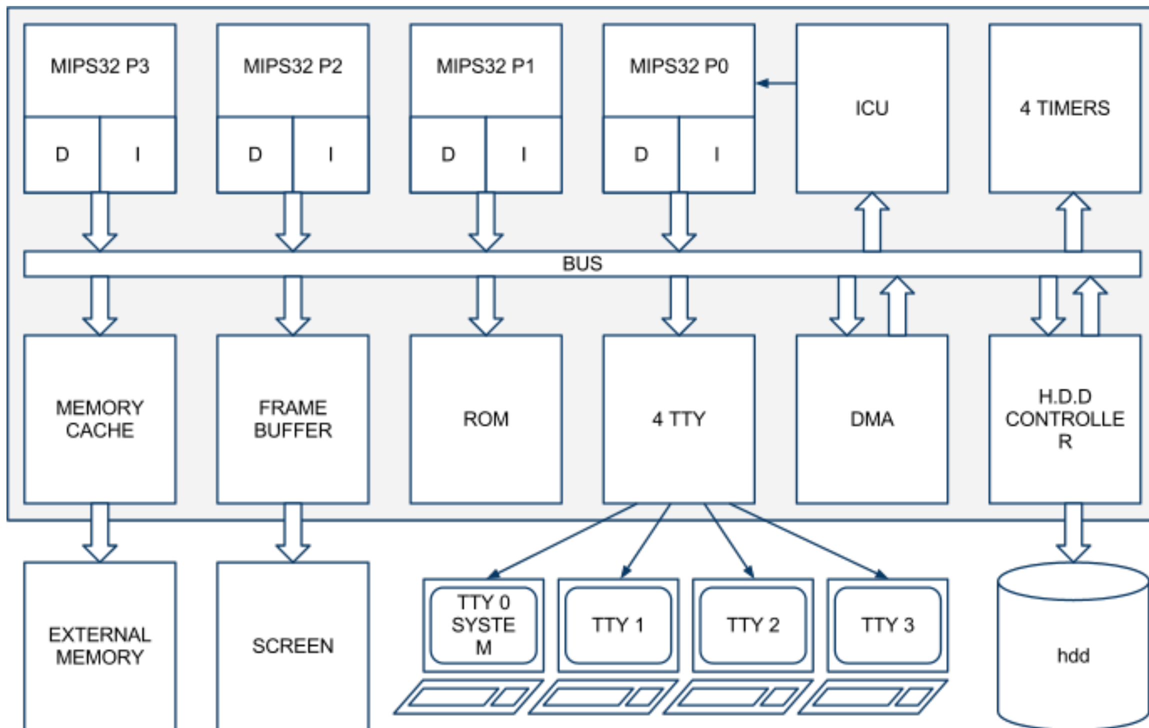
L'objectif de cette séance est de présenter en détails la programmation de la première application sur la plateforme.
Nous allons voir ainsi (pas forcément dans cet ordre) :

- le mapping mémoire
- le code de boot
- le makefile
- la synchronisation entre les processeurs
- la commande de périphérique
- les directives au compilateur en C
- l'allocation des piles de démarrage

Les programmes de ce "hello world" ne seront pas dans l'OS.
C'est seulement préparatoire.
Chaque processeur affiche "Hello World" sur "son" TTY

  - entièrement en assembleur
  - en assembleur et en C

La plateforme matérielle



description de la mémoire

fichier : segmentation.h

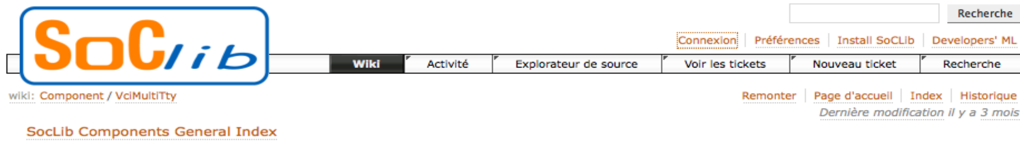
```
// ------- Devce  mapped segments

#define    TIMER_BASE  0xd3200000
#define    TIMER_SIZE    0x00000080

#define    ICU_BASE      0xd2200000
#define    ICU_SIZE       0x00000020

#define    DMA_BASE     0xd1200000
#define    DMA_SIZE      0x00000014

#define    TTY_BASE     0xd0200000
#define    TTY_SIZE      0x00000040

#define    BD_BASE       0xd5200000
#define    BD_SIZE        0x20

#define    FB_XSIZE      512
#define    FB_YSIZE      512
#define    FB_BASE       0x52200000
#define    FB_SIZE        FB_XSIZE*FB_YSIZE*2
```

```
// ------- ROM mapped segments

#define    KTEXT_LMA_BASE 0xbf800000
#define    KTEXT_LMA_SIZE   0x00020000

#define    KDATA_LMA_BASE  0xbf820000
#define    KDATA_LMA_SIZE    0x00020000

#define    BOOT_BASE           0xbfc00000
#define    BOOT_SIZE            0x00001000

// ------- RAM

#define    RAM_BASE            0x7F400000
#define    RAM_SIZE             0x01000000

// ------- Application mapped segments

#define    KTEXT_BASE         0x80000000
#define    KDATA_BASE         0x80020000
#define    KDATA_SIZE          0x003E0000

#define    USR_TEXT_BASE   RAM_BASE
#define    USR_TEXT_SIZE     0x00060000
#define    USR_DATA_BASE   \
           USR_TEXT_BASE + USR_TEXT_SIZE
#define    USR_DATA_SIZE     0x00B9F000
```

Description des périphériques

https://www.soclib.fr/trac/dev/wiki/Component

pour le TTY :
https://www.soclib.fr/trac/dev/wiki/Component/VciMultiTty



SocLib Components General Index

VciMultiTty

1) Functional Description

This VCI target is a TTY terminal controller. This hardware component controls one or several independant terminals. The number of emulated terminals is defined by the arguments in the constructor (one name per terminal).

Each terminal is acting both as a character display, and a keyboard interface. For each terminal, a specific IRQ is activated when a character entered at the keyboard is available in a buffer. IRQ is kept low as long as the buffer is not empty.

This hardware component cheks for segmentation violation, and can be used as a default target.

This component uses a TtyWrapper per terminal in order to abstract the simulated ttys. The terminal index *i* is defined by the ADDRESS[12:4] bits.

Each TTY controller contains 3 memory mapped registers:

- `TTY_WRITE`

This 8 bits pseudo-register is write only. Any write request will interpret the 8 LSB bits of the WDATA field as an ASCII character, and this character will be displayed on the addressed terminal.

- `TTY_STATUS`

This Boolean status register is read-only. A read request returns the zero value if there is no pending character. It returns a non zero value if there is a pending character in the keyboard buffer.

- `TTY_READ`

Description des périphériques

Sur la page de description du périphérique on trouve :

The file source:trunk/soclib/soclib/module/connectivity_component/vci_multi_tty/include/soclib/tty.h defines `TTY_WRITE`, `TTY_STATUS`, `TTY_READ` and `TTY_SPAN`.

fichier : devices.h

```
#ifndef _DEVICES_H_
#define _DEVICES_H_

/* TTY mapped registers offset */
#define TTY_SPAN        4
#define TTY_WRITE_REG   0
#define TTY_STATUS_REG  1
#define TTY_READ_REG    2

#endif
```

Description de la mémoire pour le linker
fichier kldscript.h

```
#include "segmentation.h"

MEMORY
{
  boot : ORIGIN = BOOT_BASE, LENGTH = BOOT_SIZE
}

SECTIONS
{
 .boot : { *(.boot) } > boot
}
```


Description de la mémoire pour le linker
fichier kldscript.h

avant

```
#include "segmentation.h"
MEMORY
{
  boot : ORIGIN = BOOT_BASE, LENGTH = BOOT_SIZE
}
SECTIONS
{
 .boot : { *(.boot) } > boot
}
```

maintenant

```
#include "segmentation.h"
MEMORY
{
  boot   : ORIGIN = BOOT_BASE, LENGTH = BOOT_SIZE
}
SECTIONS
{
 .boot : { *(.boot) *(.text) *(.boot.*) *(.data) *(.rodata*)  } > boot
}
```

le code de boot

fichier : boot.S

```
#include <devices.h>
#include <segmentation.h>

.section .boot ,"ax",@progbits
.ent    boot
.align  2

boot:
    li   $26,  0
    mtc0  $26,  $12  # Status Register
    mfc0  $4,   $0
    la    $5,   str
    j     tty_puts
```

```
//  Function name : fputs
//  Arguments     : $4 <-> processor id, //              $5 <->
@str
//  Description   : print string str
//  ------------------------------------
tty_puts:
    la    $26,   TTY_BASE
    li    $27,   TTY_SPAN * 4
    multu  $4,    $27
    mflo   $27
    addu   $26,   $26,   $27
loop:
    lbu    $27,   0($5)
    beqz   $27,   deadLoop
    sw     $27,   (TTY_WRITE_REG)($26)
    addiu  $5,    $5,    1
    j        loop
deadLoop:
    j      deadLoop

str:   .asciiz "hello world"

.end boot
```

Signification des drapeaux

Section Flag Characters	
Flag Characters	**Description**
`w`	Write access allowed.
`a`	Section is allocated in memory.
`x`	Section contains executable instructions.
`s`	Section contains "short" data.
`o`	Section adds ordering requirement. The '`o`' flag is only for `ELF` (Unix*) files.

Section Types	
Section Type	**Description**
`"progbits"`	Sections with initialized data or code.
`"nobits"`	Sections with uninitialized data (`bss`).
`"comdat"`	`COMDAT` sections, Windows NT specific. See Windows NT (COFF32) Specific Section Flag Operands.
`"note"`	Note sections.

registre status

dans MIPS_vol3 p. 53

contenu des 16 bits de poids faible du registre **SR**:

IM[7:0]	0	0	0	UM	0	ERL	EXL	IE

Cette version du processeur MIP32 n'utilise que 12 bits du registre SR :

IE : Interrupt Enable
EXL : Exception Level
ERL : Reset Level
UM : User Mode
IM[7:0] : Masques individuels pour les six lignes d'interruption matérielles (bits IM[7:2]) et pour les 2 interruptions logicielles (bits IM[1:0])

registre status

IM[7:0]	0	0	0	UM	0	ERL	EXL	IE

- Le processeur a le droit d'accéder aux ressources protégées (registres du CP0, et adresses mémoires > 0x7FFFFFFF)
  si et seulement si
  le bit UM vaut 0, ou si l'un des deux bits ERL et EXL vaut 1.
- Les interruptions sont autorisées
  si et seulement si
  le bit IE vaut 1, et si les deux bits ERL et EXL valent 00, et si le bit correspondant de IM vaut 1.
- Les trois types d'événements qui déclenchent le branchement au GIET (interruptions, exceptions et appels système) forcent le bit EXL à 1, ce qui masque les interruptions et autorise l'accès aux ressources protégées.
- L'activation du signal RESET qui force le branchement au Boot-Loader force le bit ERL à 1, ce qui masque les interruptions et autorise l'accès aux ressources protégées.
- L'instruction ERET force le bit EXL à 0.

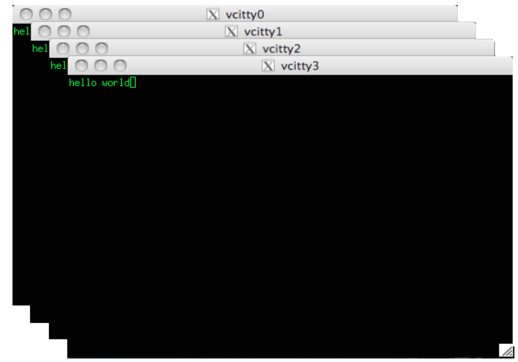
Makefile

```
INCLUDES    = segmentation.h
KERNEL      = kernel-soclib.bin
CSRC        =
SSRC        = boot.S
KOBJS       = $(SSRC:%.S=%.o) $(CSRC:%.c=%.o)

.PHONY: clean realclean

# CC tools and parameters
#------------------------------------------------------------------------------
BUILD_DIR   :=$(shell pwd)
CCTOOLS     ?=
CPU         = mipsel
CC          = $(CCTOOLS)/bin/$(CPU)-unknown-elf-gcc
LD          = $(CCTOOLS)/bin/$(CPU)-unknown-elf-ld
OD          = $(CCTOOLS)/bin/$(CPU)-unknown-elf-objdump
SIMUL       = ../../bin/simulation.x

CFLAGS      = -fno-builtin -I. -fomit-frame-pointer -O3 -G0 -Wall -Werror -mips32r2
LDFLAGS     =
TRASH       = /dev/null||true

# Kernel building rules
#------------------------------------------------------------------------------
all: depend $(KERNEL)
```

Makefile

```
$(KERNEL): $(KOBJS) kldscript
	$(LD) -o $@ $(KOBJS) $(LDFLAGS) -T$(BUILD_DIR)/kldscript
	$(OD) $@ -D > $@.dump

kldscript : kldscript.h segmentation.h
	cpp $< | egrep -v "#|//" | grep . > $@

clean:
	$(RM) vcitty* *.bak *.o *.pdf *~ kldscript *.dump $(KERNEL) 2> $(TRASH)

simul: $(KERNEL)
	$(SIMUL)

pdf:
	a2ps -1 --medium=A4 --file-align=fill -o - -l100 \
		Readme.txt Makefile *.h $(SSRC) $(CSRC) |\
	ps2pdf -sPAPERSIZE=a4 - `basename $$PWD`.pdf

#------------------------------------------------------------------------------
%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@
%.o: %.S
	$(CC) $(CFLAGS) -c $< -o $@

depend: ; makedepend -I. $(CSRC)
```

Compilation & Exécution




```
> make
makedepend -I.
/usr/local/cctools/bin/mipsel-unknown-elf-gcc -fno-builtin -I. -fomit-frame-pointer -O3 -G0 -Wall -Werror -
mips32r2 -c boot.S -o boot.o
cpp kldscript.h | egrep -v "#|//" | grep . > kldscript
/usr/local/cctools/bin/mipsel-unknown-elf-ld -o kernel-soclib.bin boot.o   -
T/Users/franck/Enseig/uem1_os/ose/tp1/e0/kldscript
/usr/local/cctools/bin/mipsel-unknown-elf-objdump kernel-soclib.bin -D > kernel-soclib.bin.dump

> make simul
../../bin/simulation.x

            SystemC 2.2.0 --- Nov  9 2009 17:32:16
        Copyright (c) 1996-2006 by all Contributors
                   ALL RIGHTS RESERVED
Mapping table: ad:(8) id:(8) cacheability mask: 0xf0000000
 &lt;Segment "boot": @0x0xbfc00000, 0x0x1000 bytes, (0), cached&gt;
 &lt;Segment "ktext": @0x0xbf800000, 0x0x20000 bytes, (0), cached&gt;
 &lt;Segment "kdata": @0x0xbf820000, 0x0x20000 bytes, (0), cached&gt;
 ....
```

Ajout de code C / boot

La différence est qu'il va falloir réserver une pile pour chaque processeur

```
#include <segmentation.h>
#include <config.h>

#define STACK_SIZE  CONFIG_BOOT_STACK_SIZE

.section  .boot ,"ax",@progbits
.extern    __do_init
.ent        boot
.align      2

boot:
   li      $26,    0
   mtc0    $26,    $12                # Status Register
   mfc0    $4,     $0                 # CPU_ID
   la      $27,    RAM_BASE+RAM_SIZE   # top memory
   li      $26,    (STACK_SIZE)
   mult    $4,     $26
   mflo    $29
   subu    $29,    $29,    $27        # $29 <= TOP - procid * STACK_SIZE
   addiu   $29,    $29,    -1*4       # for __do_init argument
   la      $26,    __do_init
   jr      $26

.end boot
```

Ajout de code C / __do_init

que se passerait-il si on n'utilisait pas de pile séparée ?

```
#include <segmentation.h>
#include <config.h>
#include <devices.h>

void tty_puts(unsigned tty_id, char *buffer)
{
   unsigned volatile *tty_base = (unsigned *) TTY_BASE + (tty_id * TTY_SPAN);
   while (*buffer)
      tty_base[TTY_WRITE_REG] = *buffer++;
}

void __do_init(unsigned cpu_id)
{
   tty_puts(cpu_id, "Hello World!\n");
   while (1);
}
```

Ajout de code C / kldscript

code objet issu du code C

```
Disassembly of section .text:

00000000 <tty_puts>:
   0:   80a20000    lb    v0,0(a1)
   4:   00042100    sll   a0,a0,0x4
   8:   3c03d020    lui   v1,0xd020
   c:   10400006    beqz  v0,28 <tty_puts+0x28>
...
  28:   03e00008    jr    ra
  2c:   00000000    nop

00000030 <__do_init>:
  30:   3c02d020    lui   v0,0xd020
  34:   00042100    sll   a0,a0,0x4
  38:   3c030000    lui   v1,0x0
...
  5c:   08000017    j     5c <__do_init+0x2c>
  60:   00000000    nop

Disassembly of section .reginfo:

00000000 <.reginfo>:
   0:   8000003c    lb    zero,60(zero)
   ...
```

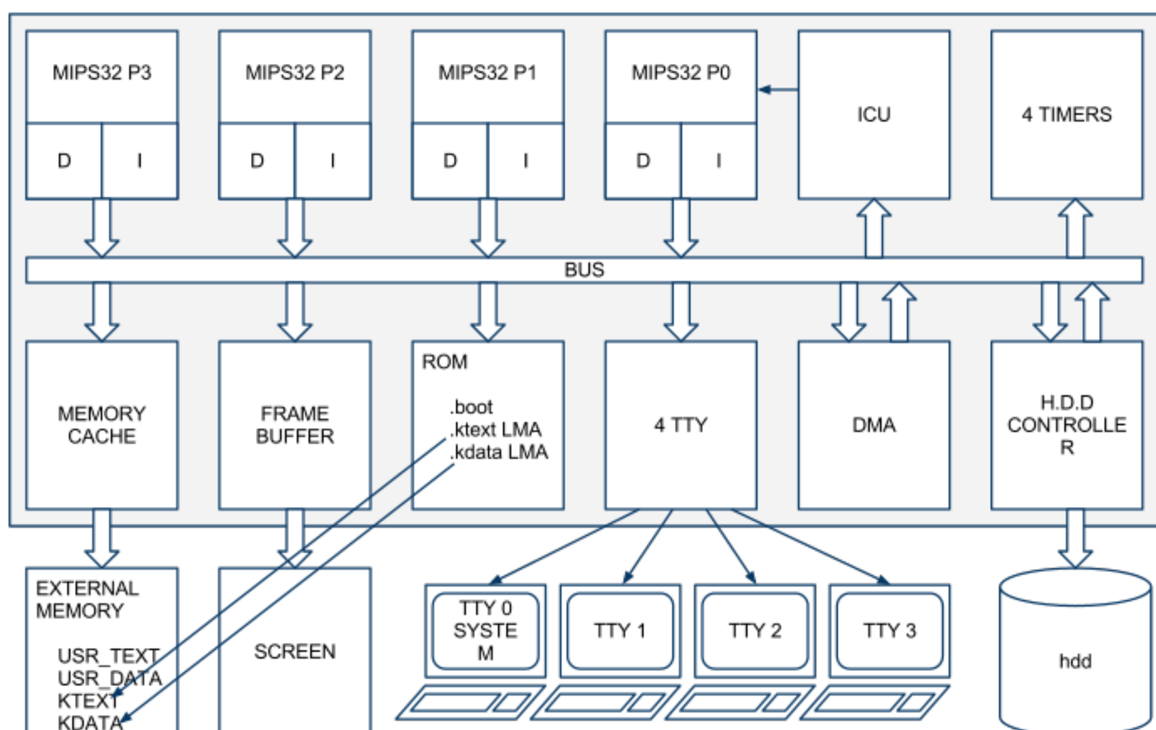
```
Disassembly of section .pdr:

00000000 <.pdr>:
   ...
  18:   0000001d    0x1d
  1c:   0000001f    0x1f
   ...
  38:   0000001d    0x1d
  3c:   0000001f    0x1f

Disassembly of section .rodata.str1.4:

00000000 <$LC0>:
   0:   6c6c6548    0x6c6c6548
   4:   6f57206f    0x6f57206f
   8:   21646c72    addi   a0,t3,27762
   c:   0000000a    movz   zero,zero,zero
```

Boot Loader

Déplacement du code système vers la RAM

MI074 -3

Action du bootloader



Placement du code et des données

Le code et les données doivent être liés (ld) pour pouvoir être exécutés dans la ram, mais ils doivent être placés dans la rom par le loader.

On utilise le LMA (Load Memory Address)

```
#include "segmentation.h"
MEMORY
{
  boot   : ORIGIN = BOOT_BASE,  LENGTH = BOOT_SIZE
  ktext  : ORIGIN = KTEXT_BASE, LENGTH = KTEXT_LMA_SIZE
  kdata  : ORIGIN = KDATA_BASE, LENGTH = KDATA_LMA_SIZE
}
SECTIONS
{
 .boot : { *(.boot) *(.boot.*) } > boot
 .ktext : AT ( KTEXT_LMA_BASE ) { *(.kentry) *(.text) __ktext_end = .; } > ktext
 .kdata : AT ( KDATA_LMA_BASE ) { *(.data*) *(.rodata*) *(.bss*) *(COMMON*)
                         *(.scommon*) __kdata_end = .; } > kdata
}
```

Le Bootloader (synchro)

Un processeur et un seul (le CPU0) doit déplacer le système.
Les autres processeurs "attendent" qu'une variable globale change.
La variable globale est déclarée dans un .c

- Quelle est l'adresse de cette variable ?

```
 .extern boot_signal   `
 la     $26,   boot_signal
 sw      $0,    ($26)
 bne    $16,   $0, boot_wait
 la     $26,   __boot_loader
 jalr   $26
```

- Réponse:
  c'est une variable de la RAM qui est initialisée à 0
  et qui vaudra le nombre de processeur à la fin du __boot_loader

code de boot complet

```
#include <segmentation.h>
#include <config.h>

#define STACK_SIZE  CONFIG_BOOT_STACK_SIZE

.section .boot ,"ax",@progbits
.extern boot_signal
.extern __boot_loader
.extern __do_init

.ent    boot
.align  2

boot:
   li     $26,    0
   mtc0   $26,    $12    # Status Reg
   mfc0   $16,    $0     # CPU_ID
   la     $27,    RAM_BASE+RAM_SIZE # top
   li     $26,    (STACK_SIZE)
   mult   $16,    $26
   mflo   $29
   subu   $29,    $29,    $27
   addiu  $29,    $29,    -1*4

# for __do_init/__boot_loader  argument
   la    $26,    boot_signal
   sw    $0,     ($26)
   # goto to boot_wait if procid != 0
   bne   $16,    $0, boot_wait
   la    $26,    __boot_loader
   jalr  $26
call_do_init:
   or    $4,     $0,     $16
   la    $26,    __do_init
   jr    $26

boot_wait:
   lw    $27,    0($26)
   sub   $5,     $27,    $16
   bgez  $5,     call_do_init
   j     boot_wait

.end boot
```

Fonction __boot_loader (__attribute__)

On doit placer cette fonction dans la section .boot :

il suffit de la déclarer avant sa définition avec un attribut section:

void __boot_loader(void) **__attribute__ ((section(".boot")));**

Les __attribute__ s'utilisent pour les fonctions ou les variables:

http://gcc.gnu.org/onlinedocs/gcc-3.3.1/gcc/Variable-Attributes.html#Variable Attributes
http://gcc.gnu.org/onlinedocs/gcc-3.3.1/gcc/Variable-Attributes.html#Function Attributes

Les autres (extrait):

- aligned(*b*) : variable alignée sur une frontière de *b* bytes
- paked : variable "tassée" pour occuper le moins de place
- noinline : fonction à ne jamais inliner
- noreturn : fonction dont on ne sort jamais (sauf exit)

Fonction __boot_loader (var ldscript)

On veut copier une partie de la rom vers la mémoire.

- On connait les adresses de début, mais les adresses de fin dépendent du remplissage.
- Les adresses de fin sont __ktext_end et __kdata_end

Pour récupérer ces adresses dans le code C:

- extern unsigned __ktext_end; // on déclare le symbole
- (unsigned) &__ktext_end // valeur typée du symbole

Fonction __boot_loader (le code)

```
#include <segmentation.h>
#include <config.h>

unsigned boot_signal;

void __boot_loader(void) __attribute__ ((section(".boot")));

void __boot_loader()
{
   unsigned i;
   unsigned size;
   extern unsigned __ktext_end;
   extern unsigned __kdata_end;

   size = ((unsigned) &__ktext_end - KTEXT_BASE) >> 2;
   for (i = 0; i < size; i++)
      *((unsigned *) KTEXT_BASE + i) = *((unsigned *) KTEXT_LMA_BASE + i);

   size = ((unsigned) &__kdata_end - KDATA_BASE) >> 2;
   for (i = 0; i < size; i++)
      *((unsigned *) KDATA_BASE + i) = *((unsigned *) KDATA_LMA_BASE + i);

   boot_signal = CONFIG_CPU_NR;
}
```

Insertion de code assembleur

On peut inclure du code assembleur dans du code C:

http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

```
asm volatile (
  assembler template
  : output operands          /* optional */
  : input operands           /* optional */
  : list of clobbered registers /* optional */
);
```

- assembler template:
  une chaine de caractères, les instructions sont séparées par \n.
- output et input operands:
  permet d'associer les variables et les registres
- registers:
  permet d'informer les compilo des registres modifiés

insertion de code assembleur

Dans le code assembleur, les registres remplacés par des variables sont notés par %0, %1, etc et ils seront associés aux variables suivant leur index.

```
static inline bool_t cpu_spinlock_trylock(uint_t * lock)
{
   register bool_t state = 1;  // preload 1 as return value (hyp lock is busy)
   asm volatile (
              "lw    $2, (%1)  \n"  // load lock value in $2
              "bnez  $2, 20    \n"  // forgive whenever lock is busy
              "ll    $2, (%1)  \n"  // load lock value in $2 and try
                                    // to register in memory
              "bnez  $2, 12    \n"  // forgive whenever lock is busy
              "sc    %0, (%1)  \n"  // sc returns in %0 <- 1 if free, 0 if busy
              "xori  %0, %0, 1 \n"  // trylock returns 0 if free, 1 if busy
              : "=r" (state)
              : "r" (lock)
              : "$2");
   return state;
}
```

affichage de la date de démarrage

```
void __do_init(unsigned cpu_id)
{
   unsigned register val;
   asm volatile ("mfc0 %0, $9\n" : "=r" (val));

   tty_puts(cpu_id, "Hello World at ");
   tty_putx(cpu_id, val);
   tty_puts(cpu_id, " !\n");
   while (1);
}
```

La suite en TME...