# HAL - CPU
# part 2

MI074 - 5

# HAL CPU : services

**Registres spéciaux**
numéro de proc, timestamp

**Irq du CPU**
masquage, démasquage.

**Opérations atomiques et lock**
addition, trylock, lock, unlock, ...

**Caches**
invalidation de ligne de cache data

**Gestionnaires d' *"exception"***
boot et kentry

**Contexte du processeur**
création, destruction, chargement, sauvegarde.

# Boot

- Ce qui se passe avant d'entrer dans le noyau
  - Initialisation des cores
    → Tous les cores démarrent en même temps
    mais ce n'est pas une règle.
  - Chargement du noyau
    → Un des cores se charge d'aller chercher le noyau et
    de le placer dans la mémoire.
    Ce core libère les autres.
  - Tous les cores appelle la fonction **__do_init()**
    qui est l'entrée du noyau.
    → Notez que cette exécution se fait avec une pile temporaire
    qui est abandonnée lorsque le premier thread du noyau démarre

# Boot

```
#define s(a)            #a
#define v(a)            s(a)

//
----------------------------------------------------------------------------------------------------
// BOOT
//
----------------------------------------------------------------------------------------------------

#define STACK_SIZE  CONFIG_BOOT_STACK_SIZE

__asm__(//
".section   .boot ,\"ax\",@progbits                 \n"
".extern     boot_signal                            \n"
".extern     __boot_loader                          \n"
".ent        boot                                   \n"
".align      2                                      \n"
"                                                   \n"
"boot:                                              \n"
"   li      $26,    0                               \n"
"   mtc0    $26,    $12                             \n" // Status Register
"   mfc0    $16,    $0                              \n" // CPU_ID
"   la      $27,    "v(RAM_BASE+RAM_SIZE)"          \n" // top memory
"   li      $26,    ("v(STACK_SIZE)")               \n"
"   mult    $16,    $26                             \n"
"   mflo    $29                                     \n"
"   subu    $29,    $29,    $27                     \n" // $29 <= TOP - procid * STACK_SIZE
"   addiu   $29,    $29,    -1*4                    \n" // for __do_init/__boot_loade  argument
"   bne     $16,    $0,     boot_wait               \n" // goto to boot_wait if procidi=0
"   la      $26,    __boot_loader                   \n"
"   jalr    $26                                     \n"
"                                                   \n"
"call_do_init:                                      \n"
"   or      $4,     $0,     $16                     \n"
"   la      $26,    __do_init                       \n"
"   jr      $26                                     \n"
"                                                   \n"
"boot_wait:                                         \n"
"   la      $26,    boot_signal                     \n"
"   lw      $27,    0($26)                          \n"
"   sub     $5,     $27,    $16                     \n"
"   bgtz    $5,     call_do_init                    \n"
"   j       boot_wait                               \n"
"                                                   \n"
".end boot                                          \n"
);
```

# kentry

- Gestionnaire
  - exceptions : __do_exception()
    - bus erreur, instruction invalide… fautes de pages
  - interruptions du CPU : __do_interrupt()
    - périphériques directement ou pas
    - logiciel
  - appels système : __do_syscall()
    - de l'utilisateur en passant par des bibliothèques comme la libc ou la libpthread

# Usage des registres (cas du mips)

- Les registres que la fonction appelante doit sauvegarder :

```
$at        : $1
$v0-$v1    : $2-$3
$a0-$a3    : $4-$7
$t0-$t7    : $8-$15
$t8-$t9    : $24-$25
$ra        : $31
```

- Les registres que la fonction appelée doit sauvegarder :

```
$s0-$s7    : $16-$23
$sp        : $29
```

# Kentry

- Analyse de la cause puisque qu'il n'y a qu'un seul point d'entrée (registre C0_CR $13) dans cet ordre
  - syscall
    - Si c'est un **syscall** on verra plus tard mais il n'y a pas de sauvegarde des registres parce que c'est un appel "prévu" (synchrone)
    - il faudra autoriser les interruptions
  - interruption
    - il faut sauver dans la pile tous les registres temporaires (cf après) (non sauvés par gcc lors de l'exécution d'une fonction)
    - c'est __do_interrupt() qui analyse la source de l'interruption grâce aux fonctions de l'API HAL-CPU
  - exception
    - il faut sauver dans la pile tous les registres restants afin d'avoir l'image complète du CPU
    - appel de __do_exception()
    - dans notre cas on ne gère pas le retour mais ce serait possible pour les instructions illégales

# kentry

```
#define     AT          0
#define     V0          1
#define     V1          2
#define     A0          3
#define     A1          4
#define     A2          5
#define     A3          6
#define     T0          7
#define     T1          8
#define     T2          9
#define     T3          10
#define     T4          11
....
#define NBA 1 // number of arguments of called functions

_asm__(
".section   .kentry,\"ax\",@progbits                \n"
".ent        kentry                                 \n"
".set        noat                                   \n"
".org        0x180                                  \n"
"                                                   \n"
"kentry:                                            \n"
"   mfc0    $26,    $13                             \n" // read CR
"   andi    $26,    $26,    0x3C                    \n" // apply cause mask
"   li      $27,    0x20                            \n" // syscall code
"   bne     $26,    $27,    not_syscall             \n" // that is not a syscall
"                                                   \n"
"syscall:                                           \n"
"   nop                                             \n"
"                                                   \n"
"not_syscall:                                       \n"
"   addiu   $29,    $29, -(\"v(SAVE_REG_NB+NBA)\")*4 \n" // saved regs + function args
"   sw      $1,     (\"v(NBA+AT)\")*4($29)          \n" // save temporary registers
"   sw      $2,     (\"v(NBA+V0)\")*4($29)          \n"
"...                                                \n"
"   sw      $28,    (\"v(NBA+GP)\")*4($29)          \n"
"   sw      $31,    (\"v(NBA+RA)\")*4($29)          \n"
"   mflo    $1                                      \n"
"   mfhi    $2                                      \n"
"   mfc0    $3,     $14                             \n"
"   sw      $1,     (\"v(NBA+LO)\")*4($29)          \n" // EPC Exception Program Counter
"   sw      $2,     (\"v(NBA+HI)\")*4($29)          \n"
"   sw      $3,     (\"v(NBA+EPC)\")*4($29)         \n"
"                                                   \n"
"interrupt:                                         \n"
"   la      $27,    __do_interrupt                  \n" // get __do_interrupt address
"   bne     $26,    $0, exception                   \n" // compare CR to 0
"   mfc0    $4,     $0                              \n" // CPU_ID, 1th arg
"   jalr    $27                                     \n" // __do_interrupt(CPUID)
"                                                   \n"
"return_from_interrupt:                             \n"
"   lw      $3,     (\"v(NBA+EPC)\")*4($29)         \n" // restore temporary registers
"   lw      $2,     (\"v(NBA+HI)\")*4($29)          \n"
"   lw      $1,     (\"v(NBA+LO)\")*4($29)          \n"
"   mtc0    $3,     $14                             \n"
"   mthi    $2                                      \n"
"   mtlo    $1                                      \n"
"   lw      $31,    (\"v(NBA+RA)\")*4($29)          \n"
"   lw      $28,    (\"v(NBA+GP)\")*4($29)          \n"
"...                                                \n"
"   lw      $2,     (\"v(NBA+V0)\")*4($29)          \n"
"   lw      $1,     (\"v(NBA+AT)\")*4($29)          \n"
"   addiu   $29,    $29, (\"v(SAVE_REG_NB+NBA)\")*4  \n"
"   eret                                            \n"
"                                                   \n"
"exception:                                         \n"
"   mfc0    $4,     $0                              \n" // CPU identifier
"   mfc0    $5,     $8                              \n" // BAR Bad Address Regsiter
"   mfc0    $6,     $9                              \n" // TSC Time Stamp Counter
"   mfc0    $7,     $12                             \n" // SR  Status Register
"   mfc0    $8,     $13                             \n" // CR  Cause Register
"   addiu   $9,     $29,(\"v(SAVE_REG_NB+NBA)\")*4   \n"
"   sw      $4,     (\"v(NBA+CPU)\")*4($29)         \n"
"   sw      $5,     (\"v(NBA+BAR)\")*4($29)         \n"
"   sw      $6,     (\"v(NBA+TSC)\")*4($29)         \n"
"   sw      $7,     (\"v(NBA+SR)\")*4($29)          \n"
"...                                                \n"
"   sw      $30,    (\"v(NBA+S8)\")*4($29)          \n"
"   sw      $26,    (\"v(NBA+K0)\")*4($29)          \n"
"   sw      $27,    (\"v(NBA+K1)\")*4($29)          \n"
"                                                   \n"
"   addiu   $4,     $29,    \"v(NBA)\"*4            \n" // regs_table
"   la      $27,    __do_exception                  \n" // __do_exception(regs_table)
"   jr      $27                                     \n"
"                                                   \n"
"   .set at                                         \n"
"   .end kentry                                     \n"
);


__attribute__((used)) void __do_interrupt(unsigned cpu_id) {
        ...
}


__attribute__((used)) static void __do_exception(unsigned *reg_table) {
       ...
       while (1);
}
```

# contexte du CPU

Ensemble des informations qui permettent de définir un contexte (un état) du CPU.

Il y a deux types d'informations:

1. celles qui servent au démarrage du thread
   - p. ex. adresse de la fonction d'entrée
2. celles qui servent en régime stationnaire.
   - p. ex. table de stockage des registres

```
#define CONTEXT_SIZE _TO_BE_DEFINED_       // context table size
extern void cpu_context_init
        (unsigned ctx[],
         unsigned mode_usr,
         unsigned stack_ptr,
         unsigned entry_func,
         unsigned exit_func,
         unsigned arg1);

extern unsigned cpu_context_save  (unsigned *ctx);
extern void cpu_context_restore   (unsigned *ctx, unsigned val);
```

# 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 ligne d'nterruption matérielles (bits IM[7:2]) et pour les 2 interruptions logicielles (bits IM[1:0])

# HAL-CPU: context

Le contexte d'un CPU est simplement un tableau de registres

```
#define s(a)            #a
#define v(a)            s(a)

// ------------------------
// CPU CONTEXT
// ------------------------
#define     S0_16       0
#define     S1_17       1
#define     S2_18       2
#define     S3_19       3
#define     S4_20       4
#define     S5_21       5
#define     S6_22       6
#define     S7_23       7
#define     SP_29       8
#define     S8_30       9
#define     RA_31       10
#define     C0_SR       11
#define     EXIT_FUNC   12
#define     ARG1        13
#define     LOADABLE    14
```

```
void cpu_context_init(unsigned ctx[],
                      unsigned mode_usr,
                      unsigned stack_ptr,
                      unsigned entry_func,
                      unsigned exit_func,
                      unsigned arg1)
{
    ctx[C0_SR]      = (mode_usr) ? 0xFC13 : 0xFC03;
    ctx[SP_29]      = stack_ptr;
    ctx[RA_31]      = entry_func;
    ctx[EXIT_FUNC]  = exit_func;
    ctx[ARG1]       = arg1;
    ctx[LOADABLE]   = 1;
    // required to be sure that all data are written
    __asm__ volatile ("sync");
}
```

- Registres persistants seulement car les registres temporaires seront sauvés dans la pile du threads qui demande le changement de contexte.
- Pointeur de pile sur la dernière case occupée
- Adresse de retour qui est au départ l'adresse de la fonction du thread
- Registre status qui sera restauré (U ou K)
- Adresse de la fonction de sortie du thread
- Argument du thread
- Booléen indiquant si un thread vient d'être créé

# La commutation de tâches

- Sauvegarder le contexte du Thread courant
- Élire un nouveau Thread à partir de la liste des Threads à l'état prêt (READY) du processeur courant, selon la politique d'ordonnancement de ce processeur.
- Restaurer le contexte du Thread élu

# Sauvegarde/restauration du *contexte*

- La sauvegarde de *contexte* est effectuée par la fonction cpu_context_save().
- Lors de la sauvegarde, la fonction cpu_context_save() écrit la valeur 0 dans le registre $2 (registre résultat) avant la sauvegarde du contexte.
- La fonction cpu_context_save() retourne la valeur 1 lorsque le thread sera restauré : la fonction cpu_context_restore()

# HAL-CPU: context

```
// unsigned cpu_context_save(struct cpu_context_s * ctx)
__asm__(
".section   .text, \"ax\", @progbits                \n"
".align     2                                       \n"
".globl     cpu_context_save                        \n" // external function
".ent       cpu_context_save                        \n"
"                                                   \n"
"cpu_context_save:                                  \n"
"   mfc0    $2,     $12                             \n" // get SR
"   sw      $16,    "v(S0_16)"*4($4)                \n" // save registers
"   sw      $17,    "v(S1_17)"*4($4)                \n"
"   sw      $18,    "v(S2_18)"*4($4)                \n"
"   sw      $19,    "v(S3_19)"*4($4)                \n"
"   sw      $20,    "v(S4_20)"*4($4)                \n"
"   sw      $21,    "v(S5_21)"*4($4)                \n"
"   sw      $22,    "v(S6_22)"*4($4)                \n"
"   sw      $23,    "v(S7_23)"*4($4)                \n"
"   sw      $29,    "v(SP_29)"*4($4)                \n"
"   sw      $30,    "v(S8_30)"*4($4)                \n"
"   sw      $31,    "v(RA_31)"*4($4)                \n"
"   sw      $2,     "v(C0_SR)"*4($4)                \n" // save SR
"   li      $2,     0                               \n" // return 0 after save
"   jr      $31                                     \n"
"                                                   \n"
".end       cpu_context_save                        \n"
);
```

# Commutation de tâches : cpu_context_save()

```
if(cpu_context_save() == 0)
{
    // restore
}
de sortie du restore
```

Quand cpu_context_save() est appelée, elle retourne 0, ce qui veut dire que les instructions du restore vont être exécutées

MAIS

Quand le contexte du thread est restauré, $2 contiendra autre chose que 0, et les instructions de retour du restore sont alors exécutées.

# HAL-CPU: context

```
__asm__(
".section   .text, \"ax\", @progbits                \n"
".align     2                                       \n"
".globl     cpu_context_restore                     \n" // external function
".ent       cpu_context_restore                     \n"
"                                                   \n"
"cpu_context_restore:                               \n"
"   lw      $27,    "v(LOADABLE)"*4($4)             \n" // get loadable flag
"   lw      $26,    "v(C0_SR)"*4($4)                \n" // get mode
"   lw      $29,    "v(SP_29)"*4($4)                \n" // get stack ptr
"   bne     $27,    $0,     cpu_context_load        \n" // if set then context_load
"   lw      $31,    "v(RA_31)"*4($4)                \n" // return to cpu_context_save
"   lw      $16,    "v(S0_16)"*4($4)                \n" // retore registers
"   lw      $17,    "v(S1_17)"*4($4)                \n"
"   lw      $18,    "v(S2_18)"*4($4)                \n"
"   lw      $19,    "v(S3_19)"*4($4)                \n"
"   lw      $20,    "v(S4_20)"*4($4)                \n"
"   lw      $21,    "v(S5_21)"*4($4)                \n"
"   lw      $22,    "v(S6_22)"*4($4)                \n"
"   lw      $23,    "v(S7_23)"*4($4)                \n"
"   lw      $30,    "v(S8_30)"*4($4)                \n"
"   mtc0    $26,    $12                             \n" // restore SR
"   move    $2,     $5                              \n"
"   jr      $31                                     \n"
"                                                   \n"
"cpu_context_load:                                  \n"
"   addiu   $29,    $29,  -8                        \n" // entry_func has one arg
"   sw      $0,     "v(LOADABLE)"*4($4)             \n" // reset loadable flag
"   lw      $27,    "v(RA_31)"*4($4)                \n" // get entry_func
"   lw      $5,     "v(EXIT_FUNC)"*4($4)            \n" // get exit_func
"   la      $31,    cpu_exit_wrap                   \n" // when entry_func returns, => exit_func($2)
"   sw      $5,     4($29)                          \n" // save exit_func address
"   lw      $4,     "v(ARG1)"*4($4)                 \n" // get thread arg1
"   mtc0    $27,    $14                             \n" // set EPC with entry_func used by eret
"   mtc0    $26,    $12                             \n" // set SR
"   eret                                            \n"
"                                                   \n"
"cpu_exit_wrap:                                     \n"
"   lw      $5,     4($29)                          \n" // get exit_func
"   addu    $4,     $0,   $2                        \n"
"   jr      $5                                      \n"
"                                                   \n"
".end       cpu_context_restore                     \n"
);
```