M1 SESI 2017-2018
Architecture Multi-Processeurs
TP6 : Interruptions vectorisées
Communications avec les périphériques

Kevin Mambu

April 5, 2018

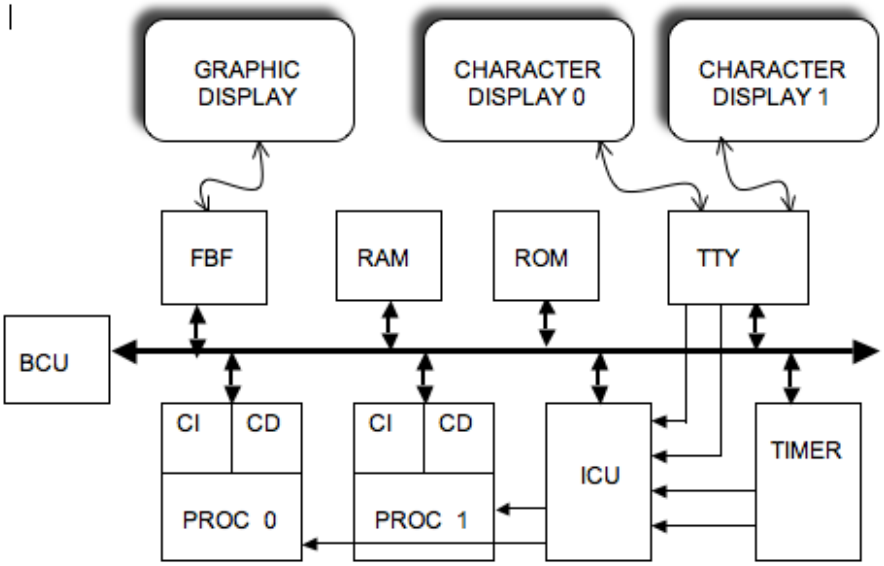
A) Objectifs

Le but de ce TP est d'analyser les mécanismes de communication par interruptions entre les périphériques et le système d'exploitation. Dans une première partie de ce TP, on illustre sur une architecture bi-processeurs le mécanisme des interruptions vectorisées en utilisant un Timer programmable, capable de générer des interrutions périodiques. Dans une seconde partie, on analyse en détail le mécanisme permettant à un programme de lire des catactères à partir d'un terminal TTY.
On dit qu'un périphérique est "mappé" en mémoire lorsqu'il possède des registres adressables par le logiciel (au moyen d'instructions de lecture ou d'écriture du type lw ou sw).

- Les registres accessibles en écriture permettent au système d'exploitation de configurer les périphériques, ou de leur envoyer des commandes.
- Les registres accessibles en lecture permettent au système d'exploitation d'obtenir des informations sur l'état du périphérique.

Pour communiquer avec le système d'exploitation, les périphériques utilisent des interruptions : Une interruption, ou IRQ (Interrupt ReQuest) est un signal booleen actif à l'état haut, qui permet à à un périphérique de "voler " quelques cycles à un processeur pour exécuter une ISR (Interrupt Service Routine). Ces ISR ont généralement pour rôle d'écrire dans des tampons mémoire appartenant au système d'exploitation. Ces tampons de communication sont spécifiques pour chaque type de périphérique.
L'architecture matérielle est identique à l'architecture matérielle définie pour le TP5, mais on utilisera seulement deux processeurs.

B) Architecture matérielle



C) Composants périphériques

Les spécifications sont en annexe

Question C1

Le composant **PibusMultiTimer** est une cible et non un maître sur le bus parce que :

- Elle n'émet pas de données sur le bus indépendamment d'une requête de la cible.
- Toute emission d'interruption (TIMER_IRQ) est en réalité à la demande du système d'exploitation.

PibusMultiTimer est d'avantage un contrôleur de timers indépendants : `ntimer` est le nombre de timers indépendants sous **PibusMultiTimer**.
Regarder l'annexe des spécifications pour les registres adressables et leurs offsets par rapport à l'adresse de base du segment SEG_TIMER_BASE

Question C2

Le composant **PibusMultiTimer** est une cible et non un maître sur le bus parce que :

- Elle n'émet pas de données sur le bus indépendamment d'une requête de la cible.
- Même si à la suite de la sortie de l'ICU, `IRQ_OUT`, un accès mémoire est fait (lecture/écriture de registres), cela est la conséquence de l'ISR associée à l'interruption et exécutée par le processeur.
- Et elle est également dépendante du système d'exploitation.

`nirq` est le nombre de signaux d'interruptions branchés sur l'ICU (cela indique par conséquence le nombre de périphériques sur l'architecture).
L'ICU du PIBUS est un contrôleur multi-canaux, `nproc` indique le nombre de processeurs reliés à l'ICU et ainsi le nombre de sorties IRQ_OUT.
L'ICU est un contrôleur reprogrammable. Lors du boot, le code du reset a parmi ses tâches d'effectuer le routage interne de l'ICU des requêtes entrantes IRQ_IN[`nirq`] vers IRQ_OUT[`nproc`]. Le système d'exploitation peut également impacter sur le routage interne.
Soit n l'identifiant du processeur au sein de l'architecture vis-à-vis de l'ICU et l'adresse de base du segment SEG_ICU_BASE, chaque processeur est associé à un sous-segment dont l'adresse de base est $SEG\_ICU\_BASE + n \times 5$. Les offsets et les fonctionnalités de chaque registe sont décrits dans l'annexe.

Question C3

L'adresse de base de l'ICU SEG_ICU_BASE doit être alignée multiple de 32×8 octets parce que chaque proceseur lui est attribué 5 registres, soit 20 octets, qu'on rehausse à 32 par souci d'alignement. Relâcher cette contrainte aurait pour conséquence de limiter le nombre de sous-canaux correctement utilisables, car l'un d'entre eux n'aurait pas de mapping mémoire suffisant pour ses registres.

Question C4

Rappelons que nous sommes sur une architecture bi-processeurs (`NPROC` $= 2$).

- IRQ_IN[0] → DMA
- IRQ_IN[1] → IOC
- IRQ_IN[2+2i] → TIMER[i]
- IRQ_IN[3+2i] → TTY[i], soit :
- IRQ_IN[3] → TTY[0]
- IRQ_IN[5] → TTY[1]
- IRQ_IN[2] → TIMER[0]
- IRQ_IN[4] → TIMER[1]

D) Lancement des tâches

Question D2

main_prime	*main_pgcd*
`0x004012dc`	`0x004013f0`

Question D3

Le flag de GCC, `freestanding` permet d'assumer que le début du programme principal ne sera pas nécessairement à l'adresse `main`. L'option `no-gpopt` demande à GCC de ne pas utiliser de pointeur global vers le segment seg_data. De ce fait, GCC utilise le mécanisme d'accès de labélisation des données, ce qui nous fait notre table de saut.

Question D4

Le programme ne peut pas notifier de l'entrée d'un caractère car le processeur n'est pas connecté à l'ICU et donc ne peut pas acquitter de l'interruption.

E) Activation du timer

Question E1

Pour se brancher à l'ISR de l'interruption concernée i, le processeur passe par le tableau interrupt_vector, qui est une table de saut, puis saute à l'adresse stockée à interrupt_vector[i].
Séquence d'appels de fonctions jusqu'à isr_timer :

- Point d'entrée au GIET
- `r27 <= Cause register`
- `r26 <= _cause_vector`
- `r26 <= &_cause_vector[XCODE]`
- `r26 <= _cause_vector[XCODE]`
- Branchement à l'adresse dans `r26` (_int_handler)

- Réservation d'espace sur la pile d'appel
- Sauvegarde de contexte (`r1-r31, HI, LO, EPC`)
- Brancchement à la fonction C `_int_demux`, de irq_handler.c
- Lecture du registre de l'ICU _ICU_IT_VECTOR
- S'il n'y a pas d'interruption active (lecture retourne 32) → retour à _irq_handler, sinon...
- `isr <= _interrupt_vector[index]` (_isr_timer)
- Branchement à la routine _isr_timer

Question E2

_isr_timer gère les deux timers connectés respectivement aux processeurs 0 et 1. Plus précisément, pour un timer TIMER[id], il acquitte l'interruption du timer puis affiche un message sur le TTY.

Question E6

Rappel, les signaux `sel_xxx` dinstinguent quel est la cible du processeur via le PIBUS à un cycle donné.

- Le processeur 0 écrit la première valeur dans le vecteur d'interruption au cycle 52 *(rappel : le vecteur d'interruptions est situé à ce moment dans la ram, car faisant partie de `sys.bin`).*
- Le registre ICU_MASK[0] est configuré au cycle 56.
- Le TIMER[0] est configuré au cycle 86.

Question E7

La première interruption du TIMER[0] est reçue au cycle 50093.

Question E8

Cycle	Évenement
100094	Levée de l'interruption `TIMER_IRQ[0]`
100114	Branchement sur le point d'entrée du GIET (0x80000180)
100123	Branchement sur `_int_handler (restore)`
100258	Branchement sur `_int_demux`
100494	Branchement sur `_isr_timer`
104348	Retour à `_int_demux`
104354	Retour à `_int_handler`
104423	Retour au programme utilisateur (`eret`)

Question E9

Le GIET est un système d'exploitation non-préemptif, en le sens qu'un processeur exécutant une routine bloquante en mode système aura malgré tout ses interruptions non-masquées, sauf si explicitement déclaré autrement.

F) Activation des interruptions TTY

Question F1

Si le mécanisme de communication par interruption n'était pas asynchrone, alors:

- Dans un cas où le programme destinataire ne serait pas exécuté par le processeur alors que l'interruption serait levée par le TTY, l'interruption resterait pendante et le contrôleur de TTY serait alors bloqué jusqu'à acquittement de l'interruption. Or, il n'y a aucun destinataire pour acquitter.

Question F2

Suite d'appels pour lire une chaîne de caractères décimaux :

- tty_getw_irq
  - sys_call(SYSCALL_TTY_READ_IRQ,...
    - syscall
  - tty_putc
    - sys_call(SYSCALL_TTY_WRITE,...
      - syscall

Question F3

La fonction _isr_tty_get est un wrapper de la fonction _isr_tty_get_indexed[0]. Si le tampon _tty_get_buf est plein au moment de l'appel de _isr_tty_get, alors la valeur précedemment contenue dans le tampon ets supprimée.

Question F4

- _tty_read_irq récupère le contenu du tampon du TTY associé au processeur appelant, puis met la bascule _tty_get_full à 0 (bascule positionnée à RESET).
- Cette fonction prend en argument l'adresse vers un tampon de reception et une longeur maximale de lecture.
- Si le tampon du TTY est vide, aucune donnée est copiée dans le tampon de reception et la fonction retourne 0.
- `tty_id = _task_context_array[(proc_id*NB_MAXTASKS + task_id)*64 + 34]` :
  - _task_context_array est la table des contextes de tâches, déclarée dans *ctx_handler.c*.
  - La macro `TASK_CTXT_SIZE` définit la taille d'un contexte de tâches, elle est égale à 64.
  - On peut deviner que 34 correspond au déplacement nécessaire pour accéder au champ tty_id d'un contexte de tâches.

Question F5

- Les caractères spéciaux traités sont :
  - `LF (line feed)` : End Of Line, associé à la touche "Enter".
  - `CR (carriage return)` : retour chariot, associé à la touche "Enter".
  - `DEL` : Suppresion de caractère, associé à la touche "Suppr"
- En cas de *Decimal String Overflow*, la chaîne de caractères est annulée, la variable recevante est mise à 0, et la fonction retourne 0.

Question F6

- L'attribut `volatile` permet d'assurer qu'une variable puisse être modifiée également depuis l'extérieur du programme. Dans notre cas, _tty_get_buf et _tty_get_full sont des registres manipulés également par le contrôleur de TTY.
- Ces registres sont rangés dans le segment `seg_tty`.
- Ce segment doit être rangés en non-cachable car ces registres peuvent être modifiés indépendamment des processeurs (modifiés par le contrôleur de TTY).

```
1  ##############################################################################
   # File : reset.s
3  # Author : Alain Greiner
   # Date : 25/12/2011
5  ##############################################################################
   #       This is an improved boot code for a bi-processor architecture.
7  # Depending on the proc_id, each processor
   #       - initializes the interrupt vector.
9  #       - initializes the ICU MASK registers.
   #       - initializes the TIMER .
11 #       - initializes the Status Register.
   #       - initializes the stack pointer.
13 #       - initializes the EPC register, and jumps to the user code.
   ##############################################################################
15
   .section .reset,"ax",@progbits
17
   .extern seg_stack_base
19 .extern seg_data_base
   .extern seg_icu_base
21
   .func reset
23 .type   reset, %function

25 reset:
        .set noreorder
27
   # get the processor id
29      mfc0  $27,  $15,  1   # get the proc_id
        andi  $27,  $27,  0x1   # no more than 2 processors
31      bne $27,  $0, proc1

33 proc0:
       # initialises interrupt vector entries for PROC[0]
35 la  $26,  _interrupt_vector
   la  $27,  _isr_timer
37 sw  $27,  8($26)      # TIMER[0] <-- _isr_timer
   la  $27,  _isr_tty_get
39 sw  $27,  12($26)     # TTY[0] <-- _isr_tty_get

41      #initializes the ICU[0] MASK register
   la  $26,  seg_icu_base    # $26 <= seg_icu_base[0]
43 addiu $27,  $0, -1    # $27 <= 0xFFFF FFFF
   sw  $27,  0xC($26)    # ICU_MASK_RESET <= 0xFFFF FFFF
45 ori $27,  $0, 0xC   # TIMER[0] and TTY[0] IRQs on
   sw  $27,  0x8($26)    # ICU_MASK_SET <= 0b0000 1100
47
       # initializes TIMER[0] PERIOD and RUNNING registers
49 la  $26,  seg_timer_base
   ori $27,  $0, 50000
51 sw  $27,  0x8($26)    # TIMER_PERIOD[0] <= 50000 cy
   sw  $27,  0x4($26)    # TIMER_RUNNING <= true
53
       # initializes stack pointer for PROC[0]
55 la  $29,  seg_stack_base
       li  $27,  0x10000     # stack size = 64K
57 addu  $29,  $29,  $27       # $29 <= seg_stack_base + 64K

59      # initializes SR register for PROC[0]
        li  $26,  0x0000FF13
61      mtc0  $26,  $12     # SR <= 0x0000FF13

63      # jump to main in user mode: main[0]
   la  $26,  seg_data_base
65      lw  $26,  0($26)      # $26 <= main[0]
   mtc0  $26,  $14     # write it in EPC register
67 eret

69 proc1:
       # initializes interrupt vector entries for PROC[1]
71 la  $26,  _interrupt_vector
   la  $27,  _isr_timer
73 sw  $27,  16($26)     # TIMER[1] <-- _isr_timer
   la  $27,  _isr_tty_get
75 sw  $27,  20($26)     # TTY[1] <-- _isr_tty_get
```

```
77          #initializes the ICU[1] MASK register
    la   $26,  seg_icu_base
79  addiu $26,  $26,  32     # $26 <= seg_icu_base[1]
    addiu $27,  $0, -1     # $27 <= 0xFFFFFFFF
81  sw   $27,  0x0C($26)    # ICU_MASK_RESET <= 0xFFFFFFFF
    ori $27,  $0, 0x30     # TIMER[1] and TTY[1] IRQs on
83  sw   $27,  0x8($26)     # ICU_MASK_SET <= 0b0011 0000

85          # initializes TIMER[1] PERIOD and RUNNING registers
    la   $26,  seg_timer_base
87  ori $27,  $0, 1
    sll $27,  $27,  4
89  or   $26,  $26,  $27    # $26 <= seg_timer_base[procid]
    li   $27,  100000
91  sw   $27,  0x8($26)     # TIMER_PERIOD[0] <= 100000 cy
    sw   $0, 0x4($26)     # TIMER_RUNNING <= false
93
          # initializes stack pointer for PROC[1]
95  la   $29,  seg_stack_base
          li   $27,  0x20000      # stack size = 128K
97  addu  $29,  $29,  $27        # $29 <= seg_stack_base + 128K

99          # initializes SR register for PROC[1]
          li   $26,  0x0000FF13
101       mtc0  $26,  $12      # SR <= 0x0000FF13

103         # jump to main in user mode: main[1]
    la   $26,  seg_data_base
105       lw   $26,  4($26)       # $26 <= main[1]
    mtc0  $26,  $14      # write it in EPC register
107 eret

109 .set reorder

111 .set reorder

113 .endfunc
    .size reset, .-reset
```

reset.s

```
////////////////////////////////////////////////////////////////////////
// File : pibus_icu.h
// Author : A. Greiner
// Date : 10/04/2010
// This program is released under the GNU Public License
// Copyright : UPMC-LIP6
////////////////////////////////////////////////////////////////////////////////
// This component implements a vectorised interrupt controler and router,
// as a PIBUS target. It concentrates up to 32 input interrupt requests
// IRQ_IN[NIRQ] and controls up to 8 output interrupt signals IRQ_OUT[NPROC].
// The NIRQ parameter defines the number of input IRQs.
// The NPROC parameter defines the number of output IRQs.
// Each OUT_IRQ[i] is the logical OR of the 32 inputs IN_IRQ[k], with
// a specific 32 bits MASK[i] depending on the output IRQ.
// These 32 bits MASK registers allow the software to route the input IRQs
// to the proper output IRQ, i.e. to the proper processor.
// IN_IRQ[i] is enabled when the corresponding mask bit is set to 1.
//
// This component takes 32 * NPROC bytes in the address space.
// and is seen as NPROC*5  memory mapped registers :
// - ICU_INT       (0x00)  (Read-Only)   returns the the 32 input IRQs.
// - ICU_MASK      (0x04)  (Read-Only)   returns the current mask value.
// - ICU_MASK_SET  (0x08)  (Write-Only)  mask <= mask | wdata.
// - ICU_MASK_RESET (0x0C)  (Write-Only)  mask <= mask & ~wdata.
// - ICU_IT_VECTOR  (0x10)  (Read-Only)   index of the smallest active IRQ.
//  (if there is no active IRQ, the returned value is 32).
//
// This component cheks address for segmentation violation,
// and can be used as a default target.
////////////////////////////////////////////////////////////////////////////////
// This component has 5 "generator" parameters :
// - sc_module_name name    : instance name
// - uint32_t   tgtid   : target index
// - pibusSegmentTable  segmap  : segment table
// - uint32_t   nirq  : number of input interrupt lines
// - uint32_t   nproc : number of output interrupt lines (default = 1)
////////////////////////////////////////////////////////////////////////////////
```

PibusIcu.desc

```
////////////////////////////////////////////////////////////////////////////
// File : pibus_multi_timer.h
// Date : 05/03/2010
// Author : A. Greiner
// It is released under the GNU Public License.
// Copyright : UPMC-LIP6
////////////////////////////////////////////////////////////////////////////
// This component is a generic timer : It contains up to 32 independant
// software controled timers.
// The timer index [i] is defined by the 5 bits ADDRESS[8:4].
// The TIMER_COUNT[i] registers used to generate periodic interrupts
// are not directly addressables.
//
// Each timer defines 4 memory mapped registers :
// - TIMER_VALUE[i] (0x0)   (read/write)
// A read request returns the value contained in TIMER_VALUE[i].
// A write request sets a new value in TIMER_VALUE[i].
// - TIMER_RUNNING[i]   (0x4) (read/write)
// A write request of a zero value gives a false value to this register.
// A write request of a non-zero value gives a true value to this register.
// When this Boolean is true, the TIMER_COUNT[i] register is
// decremented and interrupt IRQ[i] is enabled.
// - TIMER_PERIOD[i]  (0x8) (read/write)
// A write request writes a new value in the TIMER_PERIOD[i] register,
// and the new value is also  written in TIMER_COUNT[i].
// The TIMER_RUNNING[i] register is set to false.
// A read request returns the value contained in the TIMER_PERIOD[i] register.
// - TIMER_IRQ[i] (0xc) (read/write)
// A write request resets the TIMER_IRQ[i] register to false.
// A read request returns the 0 value if TIMER_IRQ[i] is false.
//
// This component cheks address for segmentation violation,
// and can be used as a default target.
////////////////////////////////////////////////////////////////////////////
// This component has 4 "constructor" parameters :
// - sc_module_name name    : instance name
// - uint32_t     tgtid    : target index.
// - PibusSegmentTable  segtab    : segment table.
// - uint32_t     ntimer    : number of independant timers
////////////////////////////////////////////////////////////////////////////
```

PibusMultiTimer.desc

```
////////////////////////////////////////////////////////////////////
// File: pibus_multi_tty.h
// Authors : Alain Greiner
// Date : 11/01/2010
// This program is released under the GNU Public License
// Copyright : UPMC-LIP6
////////////////////////////////////////////////////////////////////
// This component is a TTY controler.
// It controls up to 16 terminals emulated as XTERM windows.
// Each terminal is acting  both as a character display
// and a keyboard controler.
//
//  Each terminal is seen as 4 memory mapped registers,
// - TTY_WRITE  (0x0) (write) character to display
// - TTY_STATUS (0x4) (read)  bit0 : read buffer / bit1 : write buffer
// - TTY_READ   (0x8) (read)  the key-board character
// - TTY_CONFIG (0xc) (write) unused
//

// As a keyboard controler, it contains a TTY_READ register
// to store the character corresponding to the stroken key.
// Bit 0 of the TTY_STATUS register is 1 when TTY_READ is full.
// It must be tested by the softwre before reading a character.
// If Bit 0 of TTY_STATUS register is 0, any read request
// in register TTY_READ will return an undefined value.
// If Bit 0 of TTY_STATUS register is 1, any new stroken key will be ignored.
// Bit 0 of TTY_STATUS is forced to 1 and the IRQ_GET line is activated
// when a key is stroken.
// Bit 0 of TTY_STATUS is forced to 0, and the IRQ_GET line is
// de-activated by a read request to the TTY_READ register.
//
// As a display controler, it contains a TTY_WRITE register
// to store the character that must be diplayed.
// Bit 1 of the TTY_STATUS register is 1 when TTY_WRITE is full.
// In principle this bit must be tested before writing in TTY_WRITE register:
// If Bit 1 of TTY_STATUS register is 1, any write request to TTY_WRITE
// register will be ignored.
// Bit 1 of TTY_STATUS  is forced to 1 and the IRQ_PUT line is
// de-activated by a write request to the TTY_WRITE register.
// Bit 1 of TTY_STATUS is forced to 0, and the IRQ_PUT line is
// activated when the character is actually displayed.
//
// Implementation note : In the present implementation,
// the display buffer is supposed infinite. Therefore,
// the IRQ_PUT interrupt is not used, the Bit1 of TTY_STATUS
// is not used, and the associated flow-control mechanism.
//
// The constructor creates as many UNIX XTERM processes as
// the number of emulated terminals. It creates a PTY pseudo-terminal
// for each XTERM supporting bi-directional inter-process communication.
////////////////////////////////////////////////////////////////////
// This component has 4 "constructor" parameters :
// - sc_module_name name    : instance name
// - unsigned int tgtid     : target index
// - PibusSegmentTable  segtab    : segment table
// - unsigned int ntty      : number of terminals
////////////////////////////////////////////////////////////////////
```

PibusMultiTty.desc