SCALETTA LEZIONE SERT 10.11.2022 (E07)

1 Scrivere la funzione putu() in uart0.c per stampare un valore senza segno in decimale

idea: divido per 10, rovesciando però l'ordine. esempio: 103/10 = 10 resto 3, devo stampare prima '3', quindi metto in buffer

```
int putu(unsigned long v)
{
    char buf[11];
    int i, r, w = 0;
    if (v < 10ul) {
        w += putc(v + '0');
        return w;
    }
    i = 10;
    buf[i] = '\0';
    while (v != 0) {
        unsigned long t = v / 10;
        r = v - t * 10;
        v = t;
        buf[--i] = (char)(r + '0');
    }
    w += puts(buf + i);
    return w;
}
```

cioè va da 0 a 9, non può essere negativo, poichè sto lavorando con gli UNSIGNED

numero caratteri stampati.

semplice divisione,
recupero resto = cifra meno significativa da stampare per prima, essendo Little Endian

pre-decremento c'è "0" perchè lavoriamo con numeri tra 0 e 9, oltre userei le lettere.

stampo dall'ultima posizione in ordine cronologico, perchè non ho riempito tutto il buffer.

2 Scrivere la funzione putd() in uart0.c per stampare un valore con segno in decimale

```
int putd(long v)
{
    int w = 0;
    if (v < 0) {
        w += putc('-');
        v = -v;
    }
    w += putu(v);
    return w;
}
```

ora tratto caso con segno

caratteri stampati

stampo lui per primo

trasformo in positivo

mi riporto al caso di prima

3 Scrivere la funzione putf() in uart0.c per stampare un valore in virgola mobile

```
int putf(double v, int prec)
{
    int i, w = 0;
    if (v < 0.0) {
        w += putc('-');
        v = -v;
    }
    w += putu(v);
    w += putc('.');
    for (i = 0; i < prec; ++i) {
        v = v - (int)v;
```

'prec' è precisione in cifre decimali

caso v negativo,
metto segno negativo
e mi riconduco a caso positivo

parte intera, il double 'v' viene troncato.

parte decimale, aggiungo per ora il '.'

stampo un numero di valori dopo la virgola pari a "prec"

lascio parte frazionaria (es: 3,14 - 3 = 0.14)

```
        v = v * 10;
        w += putc('0' + (int)v);
    }
    return w;
}
```

0.14 * 10 = 1.4 di cui prelevo '1'

Led va in PANIC, è una delle eccezioni, ma quale è? DATA ABORT. Come lo so? devo cambiare il vettore data-abort, cioè dico ai led di comportarsi in modo diverso,se cambiano comportamento, allora il motivo del panic è proprio questo.

Perchè abort? perchè i numeri in virgola mobile fanno uso del coprocessore, che non è attivo. Bisogna scrivere funzione per attivarla.

4 Provando la funzione putf() si riscontra una eccezione di tipo 'Data abort' (possibile anche 'Undefined instruction')
- 4.1 Dal manuale "ARM Cortex Programming Guide" 6.1.3 apprendiamo che il coprocessore VFP deve essere esplicitamente abilitato dopo il reset
  - 4.1.1 Abilitando l'accesso ai coprocessori CP10 e CP11 tramite il Coprocessor Control Access Register
  - 4.1.2 Impostando un bit nel registro VFP FPEXC
- 4.2 Dal manuale "ARM Cortex-A8 TRM" 3.2.27 apprendiamo come si modifica il Coprocessor Control Access Register
- 4.3 Modifichiamo bbb_cpu.h aggiungendo le macro necessarie:

```
#define set_en_bit_in_fpexc() do { \
    int dummy; \
    __asm__ __volatile__ ("fmrx %0,fpexc\n\t" \
                          "orr  %0,%0,#0x40000000\n\t" \
                          "fmxr fpexc,%0" : "=r" (dummy) : :); \
} while (0)
#define read_coprocessor_access_control_register() ({ \
    u32 value; \
    __asm__ __volatile__ ("mrc p15, 0, %[reg], c1, c0, 2" : \
                          [reg] "=r" (value) : : "memory"); \
    value; })
#define write_coprocessor_access_control_register(value) \
    __asm__ __volatile__ ("mcr p15, 0, %[reg], c1, c0, 2" : : \
                          [reg] "r" (value) : "memory")
```

il codice con "fmrx" abilita l'unità di calcolo a virgola mobile (VFP) del processore, impostando il bit 30 del registro di controllo delle eccezioni a virgola mobile (FPEXC).

Registro speciale preso da manuale, LO METTO TRA LE MACRO, non hardcoded. "memory" perchè da coprocessore

- 4.4 Modifichiamo _init() aggiungendo la call alla funzione init_vfp():

```
static void init_vfp(void)
{
 u32 v = read_coprocessor_access_control_register();
 v |= (1u << 20) | (1u << 22);
 write_coprocessor_access_control_register(v);
 data_sync_barrier();
 set_en_bit_in_fpexc();
}
```

nuova macro

dettato dal manuale

5 Scrivere la funzione putcn() in uart0.c per scrivere una stringa costituita da un carattere ripetuto un dato numero di volte

```
int putcn(int ch, int n)
{
    int i;
    if (n <= 0)
        return 0;
    for (i = 0; i < n; ++i)
```

```
        putc(ch);
    return n;
}
```

```
6 Prova delle nuove funzioni:
   6.1 Aggiungere i prototipi delle nuove funzioni in comm.h
   6.2 Riscrivere la funzione banner() in main.c
```

```
static void banner(void)
{
  putcn('=',65); putnl();
  puts("SERT: System Environment for Real-Time, "
       "version 2022.11\n");
  puts("Marco Cesati, SPRG, DICII, University of Rome "
       "Tor Vergata\n");
  putcn('=',65); putnl();
}
```

Aspetti di 'contorno' per migliorare l'uso del codice, non fondamentali.

```
7 Scrivere la funzione printf() in printf.c
   7.1 Modificare la funzione banner() in main.c
```

```
static void banner(void)
{
   putcn('=', 65); putnl();
   printf(
   "SERT: System Environment for Real-Time, version %u.%x\n"
       "Marco Cesati, SPRG, DICII, University of Rome "
       "Tor Vergata\n",
       2022, 17);
   putcn('=', 65); putnl();
}
```

```
8 Gestione delle interruzioni (IRQ)
  8.1 Nel manuale della Beaglebone Black non si trovano informazioni
      specifiche sulla gestione delle interruzioni (IRQ)
  8.2 Il manuale del chip AM335 (TRM) dedica il capitolo 6 alla gestione
      delle interruzioni (60+ pagine). Vengono riassunte qui solo le
      informazioni piu' importanti
  8.3 FIQ ("fast interrupts") non sono utilizzabili su questo processore
  8.4 Un circuito "Interrupt Controller" si occupa di processare i
      segnali di interruzione provenienti dalle periferiche,
      mascherandoli e/o ordinandoli per priorita', poi produce i segnali
      di interruzione verso la CPU
  8.5 Sono gestite 128 linee di IRQ 'level-triggered' tramite diversi
      registri
       8.5.1 Registri MPU_INTC.INTC_MIRn (n=0...3) servono a mascherare
             ciascuna linea individualmente
       8.5.2 Registri MPU_INTC.INTC_ILRm (m=0...127) servono a
             selezionare per ciascuna linea IRQ o FIQ (quest'ultimo non
             disponibile su BBB), ed il valore di priorita'
             8.5.2.1 Priorita' massima associata al valore 0
             8.5.2.2 Priorita' minima associata al valore 63
```

Dobbiamo gestire le interruzioni, perchè lo scheduler opera in intervalli di tempo, e devo poter interrompere per fare context switch.
Dal manuale del chipset: "modo del dispositivo HW per segnalare alla CPU il dover cambiare il flusso di esecuzione". Ho pochi PIN per le interruzioni, mentre tante chiamate possono portare ad interruzioni. Serve INTERRUPT CONTROLLER, da "linea fisica" capisco chi la sta richiedendo. Il circuito 'intermedio' è da programmare.
Ci sono almeno 128 linee di IRQ , e servono:
- Registri MIR, sono 4, per mascherare (nascondere) linee che non mi interessano. L' I.C. risponde per PRIORITA' della linea.
- Registri ITR, per lo stato della linea senza maschera (vedo il suo stato). Sono 128.
- Pending IRQ: linee DOPO mascheramento (bit = 1 per chi ha provocato interruzione, qui ancora non so 'chi vince', cioè chi ha la priorità maggiore).
- Threshold: soglia di priorità che, se superata, porta alla 'notifica' di tale interruzioni. Se non la eccede, non viene notificata.
- SIR_IRQ_ register, che mi dice IRQ da gestire, interrupt pendente con priorità maggiore da gestire. Prima di leggerlo, chi CHIAMA INTERRUPT E' BLOCCATO. (IC segnala a CPU, IC si sblocca solo quando CPU procede nel suo lavoro).
- Control: "sblocca" la situazione nel minor tempo possibile. Conferma lo svolgimento della gestione dell'interrupt, e passo alla interruzione pendente successiva.

NB: Esistono iRQ e FastIRQ (che non vedremo). Gestiremo le interruzioni in 3 livelli:
- Procedura Assembler: salva/recupera contesto di esecuzione (da SysMode a Interrupt Mode). Non deve perdersi nulla, rimettere tutti i registri 'a posto'.
- Procedura in C: intermedia, che interagisce con l'IC e ricava l'indirizzo della ISR da eseguire.
- Funzione di alto livello in C, da eseguire a fronte dell'interrupt. Detta ISRC.

```
            8.5.2.3 In caso di parita' di priorita' vince l'interruzione
                    con il numero piu' alto
          8.5.3 Registri MPU_INTC.INTC_ITRn (n=0...3) indicano lo stato di
                ciascuna linea prima del mascheramento
          8.5.4 Registri MPU_INTC.INTC_PENDING_IRQn (n=0...3) indicano lo
                stato di ciascuna linea IRQ (non FIQ) dopo mascheramento e
                ordinamento per priorita'
          8.5.5 Registro MPU_INTC.INTC_THRESHOLD consente di indicare un
                livello di soglia delle priorita': tutti gli IRQ di
                priorita' inferiore o uguale alla soglia vengono mascherati
                automaticamente
                8.5.5.1 Priorita' massima associata al valore 0
                8.5.5.2 Priorita' minima associata al valore 63
                8.5.5.3 Valore 0xff: disabilita il meccanismo a soglia
          8.5.6 Registro MPU_INTC.INTC_SIR_IRQ indica il numero di
                interruzione pendente di priorita' massima
          8.5.7 Registro MPU_INTC.INTC_CONTROL_NEWIRQAGR serve a confermare
                l'avvenuta gestione di una interruzione e quindi a far
                selezionare la successiva interruzione pendente di priorita'
                massima
    8.6 Ricavare indirizzi di base dei vari registri INTC (in "AM335x TRM")

9 I punti salienti della gestione delle interruzioni in SERT
    9.1 Utilizzo solo interruzioni IRQ (FIQ non disponibili su BBB)
    9.2 Il modo normale di esecuzione per il sistema e' SYSTEM
         2.2.1 Tutti i "task" sono privilegiati
    9.3 I gestori delle interruzioni non fanno uso di uno stack diverso da
        quello del modo SYSTEM
          9.3.1 Il salvataggio del contesto di esecuzione all'occorrenza di
                una interruzione deve essere effettuato sullo stack SYSTEM
          9.3.2 Il modo corrente della CPU e' indicato da bit nel registro
                CPSR:
                9.3.2.1 modo SYSTEM: maschera 0x1f
                9.3.2.2 modo IRQ:    maschera 0x17
                9.3.2.3 modo FIQ:    maschera 0x11
    9.4 La gestione delle interruzioni ha tre livelli:
          9.4.1 Livello basso: procedura Assembler definita all'offset 0x18
                della tabella delle eccezioni (vedi slide 41 della lez. E03b)
                che salva e recupera il contesto di esecuzione
          9.4.2 Livello intermedio: procedura C che interagisce con l'IC e
                ricava l'indirizzo della ISR da eseguire
          9.4.3 Livello alto: ISR, ossia funzione C specifica per
                l'interruzione attivata
     9.5 Sezioni critiche implementate tramite disabilitazione delle
         interruzioni
          9.5.1 Il bit 0x80 del registro cpsr disabilita gli IRQ
          9.5.2 Il bit 0x40 del registro cpsr disabilita   i FIQ
          9.5.3 Mediante le istruzioni cpsid e cpsie

10 Scrivere il file bbb_intc.h con le definizioni dei registri
    +-------------------------------------------------------------+
    |#define INTC_BASE                0x48200000                |
    |iomemdef(INTC_SYSCONFIG,          INTC_BASE + 0x10);        |
    |iomemdef(INTC_SIR_IRQ,            INTC_BASE + 0x40);        |
    |iomemdef(INTC_CONTROL,            INTC_BASE + 0x48);        |
```

configurazione di gestione delle interruzioni

```
iomemdef(INTC_THRESHOLD,         INTC_BASE + 0x68); inizializza IC per non nascondere interruzioni.
iomemdef(INTC_ITR_BASE,          INTC_BASE + 0x80); definisco ITR[BASE] + offsets
iomemdef(INTC_MIR_BASE,          INTC_BASE + 0x84);
iomemdef(INTC_MIR_CLEAR_BASE,    INTC_BASE + 0x88); di servizio
iomemdef(INTC_MIR_SET_BASE,      INTC_BASE + 0x8c);
iomemdef(INTC_ISR_SET_BASE,      INTC_BASE + 0x90); stato registro pre-mascheramento
iomemdef(INTC_ISR_CLEAR_BASE,    INTC_BASE + 0x94);
iomemdef(INTC_PENDING_IRQ_BASE, INTC_BASE + 0x98); linee post mascheramento
iomemdef(INTC_ILR_BASE,          INTC_BASE + 0x100);
#define NUM_IRQ_LINES   128
#define NEWIRQAGR       0x1  bit per settare IC

#define irq_enable() \
    __asm__ __volatile__("cpsie i" ::: "memory");

#define irq_disable() \
    __asm__ __volatile__("cpsid i" ::: "memory");
```

10.1 Per le funzioni irq_enable() e irq_disable(), viene usata l'istruzione cps (solo in architettura ARMv7)
10.1.1 Su architetture precedenti occorre leggere, modificare e riscrivere il registro CPSR utilizzando le istr. msr e mrs
10.1.2 Per approfondimenti su cps vedere il manuale "ARM Architecture Reference Manual ARMv7-A and ARMv7-R" (issue C.c, 20 May 2014), B9.3.2

11 Modificare init.c per inizializzare l'interrupt controller:
11.1 Scrivere la funzione init_intc():

```
static void init_intc(void) all'inizio maschero linee, nessuna può usare le interruzioni.
{
    iomem(INTC_MIR_SET_BASE + 0) = 0xffffffffUL; Nessuna linea MIR può generare interruzioni.
    iomem(INTC_MIR_SET_BASE + 8) = 0xffffffffUL;
    iomem(INTC_MIR_SET_BASE + 16) = 0xffffffffUL;
    iomem(INTC_MIR_SET_BASE + 24) = 0xffffffffUL;
    iomem(INTC_THRESHOLD) = 0xff; tolgo eventuali soglie. Non c'è threshold.
    irq_enable(); Ora posso abilitare interruzioni.
}
```

11.2 Aggiungere la chiamata a init_intc() in _init()
11.3 Modificare init_vectors() in init.c:

```
extern void _irq_handler(void);
vectors[14] = (u32) _irq_handler;
```

12 Scrivere la funzione Assembly _irq_handler() in irqhandler.S Presa da un Developer Pakistano, ma andando avanti dovrò adattarla alle mie esigenze.

```
    .equ NO_IRQ,     0x80
    .equ NO_FIQ,     0x40
    .equ NO_INT,     (NO_IRQ|NO_FIQ)
    .equ FIQ_MODE,   0x11
    .equ IRQ_MODE,   0x12
    .equ SYS_MODE,   0x1f
    .section .text
```

_irq_handler: gestore a basso livello avente obiettivo di salvare e recuperare il contesto di esecuzione. VEDERE SLIDE 09 dove viene spiegato meglio.


```
    .code 32
    .globl _irq_handler
_irq_handler:
    mov     r13,r0
    sub     r0,lr,#4
    mov     lr,r1
    mrs     r1,spsr
    msr     cpsr_c,#(SYS_MODE|NO_IRQ)
    stmfd   sp!,{r0,r1}
    stmfd   sp!,{r2-r3,r12,lr}
    mov     r0,sp
    sub     sp,sp,#(2*4)
    msr     cpsr_c,#(IRQ_MODE|NO_IRQ)
    stmfd   r0!,{r13,r14}
    msr     cpsr_c,#(SYS_MODE|NO_IRQ)
    ldr     r12,=_bsp_irq
    mov     lr,pc
    bx      r12
    msr     cpsr_c,#(SYS_MODE|NO_INT)
    mov     r0,sp
    add     sp,sp,#(8*4)
    msr     cpsr_c,#(IRQ_MODE|NO_INT)
    mov     sp,r0
    ldr     r0,[sp,#(7*4)]
    msr     spsr_cxsf,r0
    ldmfd   sp,{r0-r3,r12,lr}^
    nop
    ldr     lr,[sp,#(6*4)]
    movs    pc,lr
```

Questo è il riferimento al gestore di medio livello, che dovrò scrivere io.

Deve dialogare con I.C, prendere il "numeretto" della linea da gestire. Poi questa verrà gestita da altri.

13 Definire il gestore di interruzione di medio livello

13.1 Definire il tipo di dati isr_t e la macro NULL in comm.h

```
#define NULL ((void *)0)
typedef void (*isr_t)(void);
```

di tipo "isr_t", che riceve "nulla" e ritorna "nulla". E' un "tipo funzione".

13.2 Definire due vettori locali ad un file irq.c

```
static isr_t ISR[NUM_IRQ_LINES];
unsigned long irqcount[NUM_IRQ_LINES] = { 0, };
```

Numero di elementi

Numero di interruzioni PER OGNI linea.

13.2.1 ISR memorizza gli indirizzi delle procedure di gestione di ciascun IRQ

13.2.2 irqcount memorizza il numero di occorrenze di interruzioni di ciascun tipo (solo per diagnostica del sistema)

13.3 Scrivere la funzione C _bsp_irq() in irq.c

```
void _bsp_irq(void)
{
   isr_t isr;
   u32 irqno;
   /* Cancel any soft irq */
   iomem(INTC_ISR_CLEAR_BASE + 0) = 0xffffffffUL;
   iomem(INTC_ISR_CLEAR_BASE + 8) = 0xffffffffUL;
```

Gestore medio livello

Cancello "memoria" dei registri.

```
    iomem(INTC_ISR_CLEAR_BASE + 16) = 0xffffffffUL;
    iomem(INTC_ISR_CLEAR_BASE + 24) = 0xffffffffUL;
    for (;;) {  Finisce solo con break
        if (iomem(INTC_PENDING_IRQ_BASE + 0) == 0 &&
            iomem(INTC_PENDING_IRQ_BASE + 8) == 0 &&
            iomem(INTC_PENDING_IRQ_BASE + 16) == 0 &&
            iomem(INTC_PENDING_IRQ_BASE + 24) == 0)
            return;
        /* there are pending unmasked IRQs on some IC */
        /* read the (highest-priority) IRQ line number */
        irqno = iomem(INTC_SIR_IRQ);
        /* Do nothing if a spurious interrupt is detected
           (see AM335x TRM, 6.2.5) */
        if (irqno < NUM_IRQ_LINES) {
            isr = ISR[irqno];  gestore interruzione associato. Se esiste interruzione -> deve esistere il suo gestore.
            if (!isr)
                panic0();  se non c'è gestore interruzione associato, chiamo panic0, deve essere gestito!!
            /* invoke the ISR (with IRQ disabled) */
            isr();
            /* just in case the ISR has left enabled the IRQs */
            irq_disable();
            ++irqcount[irqno];
        }
        iomem(INTC_CONTROL) = NEWIRQAGR;  Sblocco dopo che ho gestito l'interruzione.
        data_sync_barrier();              la data_sync_barrier() è da manuale.
    }
}
```

Vedo se c'è qualcosa da fare. Se tutte le 128 linee sono a 0, non ho nulla da gestire, posso uscire dall'if loop.

Potrebbero esserci delle Interruzioni Spurie, cioè dei falsi positivi. Hanno irqno>=128.

Semplicemente basta non gestirle, e concentrarsi su interruzioni vere con irqno<128.

13.4 Scrivere la funzione register_isr(): Le "interrupt service Routines" devo essere registrate.

```
int register_isr(int n, isr_t func)
{                                   'func' gestisce l'interruzione.
   if (n >= NUM_IRQ_LINES) {
      printf(
          "ERROR in register_isr(): IRQ number %u is invalid\n",
             n);
      return 1;
   }
   if (ISR[n] != NULL) {  Gestore già registrato.
    printf(
         "ERROR in register_isr(): IRQ %u already registered\n",
             n);
      return 1;
   }
   ISR[n] = func;
   return 0;
}
```

13.4.1 Aggiungere il suo prototipo in comm.h

```
/*
vim: tabstop=4 softtabstop=4 expandtab list colorcolumn=74 tw=73
*/
```