Programmare con il Bus Mastering PCI-IDE

G. Lettieri

17 Aprile 2018

Proviamo a scrivere un programma che legga dall'hard-disk in modalità Bus Mastering, facendo riferimento alle specifiche del Bus Master IDE Controller (disponibili all'indirizzo `http://http://calcolatori.iet.unipi.it/deep/idems100.pdf`).

Le operazioni da eseguire sono descritte nelle specifiche, nella Sezione 3.1. Il programma risultante è illustrato nelle Figure 1, 2 e 3.

Prima ancora di procedere con le operazioni di cui sopra, però, dobbiamo sapere a che indirizzi si trova il ponte PCI-ATA. In Figura 1, alle righe 7–12, cerchiamo dunque il ponte tra i dispositivi PCI installati. Le specifiche ci dicono che i primi due byte del Class Code del ponte devono valere 0x0101 (Sezione 5.0, punto 1). Il Class Code è il campo di 3 byte all'offset 9 dello spazio di configurazione PCI (Libro II, Figura 9.3 a pag. 183). La funzione `cerca_bm()` (Figura 2, linee 42–48) cerca dunque il primo dispositivo che contenga 0x0101 nella word che si trova all'offset 10. Si noti che, per semplicità, cerchiamo solo nel bus 0. Le specifiche dicono che l'indirizzo base dei registri del ponte è controllato dalla BAR che si trova all'offset 36 (Sezione 5.0, punto 2). Inoltre, le specifiche ci dicono che i registri si trovano nello spazio di I/O agli offset 0, 2 e 4 rispetto alla base (Sezione 2.0; si noti che ci sono anche altri registri per il canale ATA "secondario", che tralasciamo). In Figura 1 leggiamo dunque la BAR di offset 36 (linea 8), azzeriamo il bit meno significativo che, essendo i registri nello spazio di I/O, valeva 1 (linea 9). In questo modo otteniamo la base, e possiamo poi ottenere gli indirizzi dei tre registri sommandovi gli offset (linee 10–12). C'è anche un'altra operazione preliminare da compiere: alle righe 13 e 14 settiamo i bit 0 e 2 del Command Register, per abilitare il ponte a rispondere alle transazioni di I/O (nel caso non fosse già abilitato a farlo, ma molto probabilmente era già stato abilitato dal BIOS) e soprattutto a operare in bus mastering (si veda Libro II, pag. 182).

Alle righe 16–18 associamo la funzione `a_bmide` al piedino 14 dell'APIC, tramite il tipo 0x60. Il driver dovrà farci sapere quando l'operazione è conclusa, ponendo a `true` la variabile globale `done` (righe 20 e 37 di Figura 2).

Alle righe 20–22 dichiariamo un po' di variabili. Vogliamo leggere `nn` settori a partire dal settore numero `lba`. Il contenuto di questi settori deve essere scritto in `vv`, che è dichiarato in Assembler (Figura 3, righe 17–19) per motivi che vedremo tra poco.

```
int main()
{
        natl base;
        natw cmd;
        natb bus = 0, dev, fun;

        cerca_bm(dev, fun);
        base = pci_read_confl(bus, dev, fun, 32);
        base &= 0xFFFFFFFE;
        iBMCMD = base;
        iBMSTR = base + 2;
        iBMDTPR = base + 4;
        cmd = pci_read_confw(bus, dev, fun, 4);
        pci_write_confw(bus, dev, fun, 4, cmd | 0x0005);

        gate_init(0x60, a_bmide);
        apic_set_VECT(14, 0x60);
        apic_set_MIRQ(14, false);

        natb work; natq ww;
        natb nn = 1;
        natl lba = 0;

        for (int i = 0; i < BUFSIZE; i++)
                vv[i] = '-';

        for (int i = 0; i < 80; i++)
                char_write(vv[i]);

        ww = reinterpret_cast<natq>(&vv[0]);
        prd[0] = static_cast<natl>(ww);
        prd[1] = 0x80000000 | ((nn * 512) & 0xFFFF);
        ww = reinterpret_cast<natq>(&prd[0]);
        outputl(static_cast<natl>(ww), iBMDTPR);
        inputb(iBMCMD, work);
        work |= 0x08;
        outputb(work, iBMCMD);
        inputb(iBMSTR, work);
        work &= 0xF9;
        outputb(work, iBMSTR);

        natb lba_0 = lba,
             lba_1 = lba >> 8,
             lba_2 = lba >> 16,
             lba_3 = lba >> 24;
        outputb(lba_0, iSNR);
        outputb(lba_1, iCNL);
        outputb(lba_2, iCNH);
        natb hnd = (lba_3 & 0x0F) | 0xE0;
        outputb(hnd, iHND);
        outputb(nn, iSCR);
        outputb(0x00, iDCR);
        outputb(0xC8, iCMD);

        inputb(iBMCMD, work);
        work |= 0x01;
        outputb(work, iBMCMD);

        while (!done)
                ;

        for (int i = 0; i < 80; i++)
                char_write(vv[i]);
        pause();

}
```

Figura 1: Parte C++, funzione `main`.

```
#include <libce.h>
#include <apic.h>

const ioaddr iBR = 0x01F0;
const ioaddr iERR = 0x01F1;
const ioaddr iSCR = 0x01F2;
const ioaddr iSNR = 0x01F3;
const ioaddr iCNL = 0x01F4;
const ioaddr iCNH = 0x01F5;
const ioaddr iHND = 0x01F6;
const ioaddr iCMD = 0x01F7;
const ioaddr iSTS = 0x01F7;
const ioaddr iDCR = 0x03F6;
const ioaddr iASR = 0x03F6;

ioaddr iBMCMD;
ioaddr iBMSTR;
ioaddr iBMDTPR;

volatile bool done = false;
extern "C" natl prd[];
extern "C" char vv[];
const natl BUFSIZE = 65536;

extern "C" void a_bmide();
extern "C" void c_bmide()
{
        natb work;

        outputb(0x0A,iDCR);
        inputb(iBMCMD, work);
        work &= 0xFE;
        outputb(work, iBMCMD);
        inputb(iBMSTR, work);
        inputb(iSTS, work);

        done = true;

        apic_send_EOI();
}

void cerca_bm(natb& dev, natb& fun)
{
        for (dev = 0; dev < 32; dev++)
                for (fun = 0; fun < 8; fun++)
                        if (pci_read_confw(0, dev, fun, 10) == 0x0101)
                                return;
}

int main()
{
        ... // Vedere Figura 1
}
```

Figura 2: Parte C++, altre definizioni.

```
#include "libce.s"
.text
.extern         c_bmide
.global         a_bmide
a_bmide:
                salva_registri
                call    c_bmide
                carica_registri
                iretq

.data
.balign 4
.global prd
prd:
        .fill 16384, 4
.balign 4
.global vv
vv:
        .fill 65536, 1
```

Figura 3: Parte Assembler.

Per far vedere che il contenuto di `vv` cambierà senza che il nostro programma vi scriva esplicitamente, lo inizializziamo preventivamente con dei caratteri "`-`". Quando l'operazione di Bus Mastering sarà conclusa (linee 59–63), dovremmo trovare che questi caratteri sono stati sostituiti con i byte letti dall'hard disk. Per rendere l'output più evidente, inizializziamo l'hard disk della macchina virtuale con un numero sufficiente di caratteri "`@`". L'hard disk è simulato dal file che si trova in `CE/share/hd.img` nella propria directory *home*. Per scrivervi 64KiB di chiocciole si può eseguire il seguente comando:

```
perl -e 'print "@"x65536' | dd of=~/CE/share/hd.img conv=notrunc
```

(Il comando prima della pipe "`|`" stampa 65536 caratteri "`@`" e il secondo li scrive nel file, a partire dall'inizio e senza cambiarne le dimensioni).

Il resto del programma segue abbastanza da vicino lo schema suggerito nella Sezione 3.1 delle specifiche:

1. Prepariamo la tabella dei PRD (righe 30–32);
2. Scriviamo l'indirizzo di partenza della tabella nel registro che nel libro abbiamo chiamato BMDTPR (linea 34);
3. Programmiamo il controllore dell'hard disk per il trasferimento in DMA (linee 42–53); la differenza con una normale operazione di lettura è solo nel comando scritto nel registro `iCMD` (riga 53); si noti che abilitiamo il controllore a inviare richieste di interruzione (linea 52);
4. Avviamo anche il ponte, ponendo a 1 lo "Start bit" nel registro che abbiamo chiamato BMCMD (la specifica lo descrive nella Sezione 2.1);
5. le azioni descritte in questo punto della specifica sono svolte dal ponte e dal controllore, noi non dobbiamo fare niente;
6. aspettiamo che arrivi l'interrupt (linee 59–60);
7. le azioni qui descritte sono svolte direttamente dal driver (linee 31–35 di Figura 2); in aggiunta, il driver disabilita ulteriori richieste di interruzione da parte del controllore (linea 30), cosa non strettamente necessaria.

Oltre alle azioni precedenti, alle linee 35–37 specifichiamo la direzione di trasferimento (bit 3 del registro BMCMD, Sezione 2.1 delle specifiche) e resettiamo i bit 1 e 2 del registro BMSTS (descritto nella Sezione 2.2), cosa che era richiesta in una precedente revisione delle specifiche.

Se compiliamo e carichiamo questo programma dovremmo vedere una riga di trattini (stampata alle linee 27–28) seguita da una linea di chiocciole (stampata alle linee 62–63).

Allineamento e confini

Alla fine della sezione 1.2 delle specifiche troviamo una nota che dice che le regioni di memoria specificate tramite i PRD non devono trovarsi a cavallo dei "confini di 64KiB". Per "confini di xB" si intendono tutti gli indirizzi che sono multipli di x byte (allineati a x byte). La nota non dice cosa succede se la regione di memoria attraversa uno di questi confini, ma sembra implicare che il ponte PCI-ATA potrebbe avere un sommatore di soli 16 bit. Il ponte deve usare un sommatore per ottenere gli altri indirizzi a cui scrivere (o leggere), dato l'indirizzo di partenza specificato nella prima Dword del PRD. Se il sommatore ha solo 16 bit e arriva, per esempio, all'indirizzo 0x1122FFFF, passerà poi all'indirizzo 0x11220000 invece che a 0x11230000. Ovviamente, se accade questo, i risultati sono catastrofici. Possiamo vedere il problema in azione provando a traferire 64KiB nel programma precedente. Ci basta modificare la linea 21 in

```
nn = BUFSIZE / 512;
```

(La costante `BUFSIZE` è definita alla linea 23 di Figura 2 e vale appunto 64KiB.) Aggiungiamo anche il seguente codice tra le righe 63 e 64:

```
for (int i = BUFSIZE - 80; i < BUFSIZE; i++)
        char_write(vv[i]);
```

In questo modo visualizziamo il contenuto degli ultimi byte del vettore `vv`, dopo che è terminata l'operazione di lettura. Ci aspettiamo di vedere un'altra linea di chiocciole, ma se lanciamo il programma vediamo invece una linea di trattini. Dove sono finiti i byte letti dall'hard disk? Possiamo scoprirlo guardando l'indirizzo assegnato dal collegatore al vettore `vv`. Per esempio, se eseguiamo il comando

```
nm | grep vv
```

nella directory in cui si trova il file `a.out`, otteniamo l'indirizzo cercato. L'indirizzo esatto può variare da computer a computer, ma in generale non sarà allineato a 64KiB (per esserlo dovrebbe avere le ultime quattro cifre pari a 0). La dimensione di `vv` è stata scelta pari a 64KiB, e dunque entra precisamente tra due confini: se non è allineato dovrà sicuramente attraversarene uno. Ciò che è accaduto, dunque, è che il ponte ha scritto in `vv` dall'inizio fino al primo confine, ma poi è tornato indietro al confine precedente, sovrascrivendo qualunque cosa vi fosse.

Possiamo osservarlo nel nostro programma. Supponiamo che l'indirizzo di `vv` sia 0x215464. Il confine *precedente* è dunque a 0x210000. Aggiungiamo il seguente codice al programma main, dopo il codice aggiunto precedentemente e prima della linea 64:

```
char *buf = reinterpret_cast<char *>(0x210000);
for (int i = 0; i < 80; i++)
        char_write(buf[i]);
```

In questo mdo trasformiamo l'indirizzo 0x210000 in un puntatore a un buffer di caratteri e mostriamo sullo schermo la prima riga. Se lanciamo il nuovo programma dovremmo ora vedere le chiocciole mancanti.

Abbiamo sostanzialmente due modi per risolvere questo problema:

- Allineare il buffer cambiando la linea 16 in Figura 3 in `.balign 65536`.
- Eseguire più trasferimenti distinti, in modo che nessuno di essi attraversi un confine.

Il primo modo è semplice, ma spreca spazio e potrebbe non essere sempre fattibile. Il secondo è sempre fattibile e possiamo anche sfruttare il fatto che il ponte può essere programmato per passare da solo da un trasferimento al successivo, preparando una sequenza di PRD. Provare a scrivere il codice che prepara i necessari PRD, dato un indirizzo di partenza p e un numero di byte l qualsiasi.