Programmare con il Bus Mastering PCI-IDE

G. Lettieri

2 Maggio 2021

Scriviamo un programma che legge dall'hard-disk in modalità Bus Mastering, facendo riferimento alle specifiche del Bus Master IDE Controller (disponibili all'indirizzo `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 1–4, che riportano gran parte dell'esempio `bm-hard-disk` di `esempiIO`.

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 8–13, 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 (si veda la dispensa sul PCI, Figura 4). La funzione `bm_find()` (Figura 2) cerca dunque il primo dispositivo che contenga 0x0101 nella word che si trova all'offset 10 (usando una funzione di supporto definita in libce). 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 2 leggiamo dunque la BAR di offset 36 (linea 24), azzeriamo il bit meno significativo che, essendo i registri nello spazio di I/O, valeva 1 (linea 25). In questo modo otteniamo la base, e possiamo poi ottenere gli indirizzi dei tre registri sommandovi gli offset (linee 26–28). C'è anche un'altra operazione preliminare da compiere: alle righe 29 e 30 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` (Figura 4) al piedino 14 dell'APIC, tramite il tipo `HD_VECT`. Lo scopo della funzione è di invocare `c_bmide()`, definita in Figura 3. Il driver dovrà farci sapere quando l'operazione è conclusa, ponendo a **`true`** la variabile globale `done` (riga 39 di Figura 3 e riga 38 di Figura 1).

Alle righe 3–5 dichiariamo un po' di variabili. Vogliamo leggere `nn` settori a partire dal settore numero `lba`. Il contenuto di questi settori deve essere scritto

```
1  int main()
2  {
3          natb nn = BUFSIZE / 512;
4          natb lba = 0;
5          natb bus = 0, dev = 0, fun = 0;
6
7          clear_screen(0x0f);
8          if (!bm_find(bus, dev, fun)) {
9                  printf("bm non trovato!\n");
10                 pause();
11                 return 0;
12         }
13         printf("PCI-ATA at %2x:%2x:%2x\n", bus, dev, fun);
14         bm_init(bus, dev, fun);
15
16         apic_set_VECT(14, HD_VECT);
17         gate_init(HD_VECT, a_bmide);
18         apic_set_MIRQ(14, false);
19
20         for (int i = 0; i < BUFSIZE; i++)
21                 vv[i] = '-';
22         printf("primi 80 caratteri di vv:\n");
23         for (int i = 0; i < 80; i++)
24                 char_write(vv[i]);
25         printf("ultimi 80 caratteri di vv:\n");
26         for (int i = BUFSIZE - 80; i < BUFSIZE; i++)
27                 char_write(vv[i]);
28
29         prd[0] = reinterpret_cast<natq>(vv);
30         prd[1] = 0x80000000 | ((nn * 512) & 0xFFFF);
31         bm_prepare(reinterpret_cast<natq>(prd), false);
32
33         hd_enable_intr();
34         hd_start_cmd(lba, nn, READ_DMA);
35         bm_start();
36
37         printf("aspetto l'interrupt...\n");
38         while (!done)
39                 ;
40         printf("primi 80 caratteri di vv:\n");
41         for (int i = 0; i < 80; i++)
42                 char_write(vv[i]);
43         printf("ultimi 80 caratteri di vv:\n");
44         for (int i = BUFSIZE - 80; i < BUFSIZE; i++)
45                 char_write(vv[i]);
46         pause();
47 }
```

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

```
1  #include <libce.h>
2  #include <apic.h>
3  #include <hd.h>
4  #include <bm.h>
5
6  ...
7
8  namespace bm {
9          ioaddr iBMCMD;
10         ioaddr iBMSTR;
11         ioaddr iBMDTPR;
12 }
13 using namespace bm;
14
15 bool bm_find(natb& bus, natb& dev, natb& fun)
16 {
17         natb code[] = { 0xff, 0x01, 0x01 };
18
19         return pci_find_class(bus, dev, fun, code);
20 }
21
22 void bm_init(natb bus, natb dev, natb fun)
23 {
24         natl base = pci_read_confl(bus, dev, fun, 0x20);
25         base &= ~0x1;
26         iBMCMD  = static_cast<ioaddr>(base + 0);
27         iBMSTR  = static_cast<ioaddr>(base + 2);
28         iBMDTPR = static_cast<ioaddr>(base + 4);
29         natw cmd = pci_read_confw(bus, dev, fun, 4);
30         pci_write_confw(bus, dev, fun, 4, cmd | 0b101);
31 }
```

Figura 2: Parte C++, inizializzazione del bus master.

```
1  void bm_prepare(natq prd, bool write)
2  {
3          outputl(prd, iBMDTPR);
4          natb work = inputb(iBMCMD);
5          if (write) {
6                  work &= ~0x8;
7          } else {
8                  work |= 0x8;
9          }
10         outputb(work, iBMCMD);
11         work = inputb(iBMSTR);
12         work |= 0x6;
13         outputb(work, iBMSTR);
14 }
15
16 void bm_start()
17 {
18         natb work = inputb(iBMCMD);
19         work |= 1;
20         outputb(work, iBMCMD);
21 }
22
23 void bm_ack()
24 {
25         natb work = inputb(iBMCMD);
26         work &= 0xFE;
27         outputb(work, iBMCMD);
28         inputb(iBMSTR);
29 }
30
31 volatile bool done = false;
32 extern char vv[];
33 const natl BUFSIZE = 65536;
34 extern natl prd[];
35
36 extern "C" void a_bmide();
37 extern "C" void c_bmide()
38 {
39         done = true;
40         bm_ack();
41         hd_ack_intr();
42         apic_send_EOI();
43 }
```

Figura 3: Parte C++, avvio e completamento dell'operazione di trasferimento.

```
1  #include "libce.s"
2  .text
3  .extern         c_bmide
4  .global         a_bmide
5  a_bmide:
6                  salva_registri
7                  call    c_bmide
8                  carica_registri
9                  iretq
10
11 .data
12 .balign 4
13 .global prd
14 prd:
15         .fill 16384, 4
16 .balign 4
17 .global vv
18 vv:
19         .fill 65536, 1
```

Figura 4: Parte Assembler.

in `vv`, che è dichiarato in Assembler (Figura 4, righe 16–19) per motivi che vedremo tra poco.

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 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 (Figura 1, righe 29–30);

2. Nella funzione `bm_prepare()` (Figura 3) scriviamo l'indirizzo di partenza della tabella nel registro che nel libro abbiamo chiamato BMDTPR (li-

nea 3); scegliamo il trasferimento *verso* la memoria (linee 4–10); azzeriamo i bit Interrupt e Error nel registro di stato (linee 11–13)[1];

3. Programmiamo il controllore dell'hard disk per il trasferimento in DMA usando la stessa funzione `hd_start_cmd()` già usata per i normali trasferimenti di lettura e scrittura. L'unica differenza è nel comand scritto nel registro CMD (riga 34 di Figura 1); si noti che abilitiamo il controllore a inviare richieste di interruzione (linea 33);

4. nella funzione `bm_start()` (Figura 3 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 38–39);

7. le azioni qui descritte sono svolte direttamente dal driver `c_bmide()`, tramite la funzione `bm_ack()` di Figura 3); in aggiunta, il driver disabilita ulteriori richieste di interruzione da parte del controllore (linea 41), cosa non strettamente necessaria.

Se compiliamo e carichiamo questo programma dovremmo vedere due righe di trattini (stampate alle linee 22–27) seguite da due righe di chiocciole (stampate alle linee 40–45).

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. 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 33 di Figura 3 e vale appunto 64KiB.) Se lanciamo il programma vediamo che questa volta, al posto della seconda riga di chiocciole, viene stampata una riga di trattini. Dove sono finiti i byte

[1] Si noti che, come scritto nelle specifiche a pagina 4, questi due bit si azzerano scrivendovi un 1.

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 la riga 39:

```
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 4 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.