1 Lezione del 08-04-25

Concluso il discorso sulla memoria virtuale, ci concentreremo nuovamente sull'hardware, nella prospettiva di approfondire l'interazione fra nucleo e dispositivi di I/O.

1.1 Bus PCI

Il bus **PCI**, che sta per *Peripheral Component Interconnect*, è uno standard per bus sviluppato da IBM per consentire l'espansione dei loro calcolatori attraverso schede apposite, supportando quindi una cosiddetta *architettura aperta*.

Possiamo quindi immaginare che ogni scheda di espansione sia provvista dei suoi registri, delle sue interruzioni, ecc... che non devono sovrapporsi con quelli di altre schede. Storicamente, questo rappresentava un problema, in quanto potevano crearsi *conflitti* fra più schede.

Inoltre, un problema era rappresentato dai *driver*, in quanto non esisteva un modo standardizzato per rilevare se una certa scheda era installata o no, e quindi se si poteva usare un certo driver.

I produttori stabilirono quindi una sorta di standard *de facto*, che abbiamo già nominato: l'**ISA** (*Industry Standard Architecture*).

L'idea fondamentale è che la scheda non può avere registri fissi, ma deve essere programmabile in questo dal calcolatore. Inoltre, deve esistere una qualche modalità per rilevare le schede correntemente installate nel sistema.

Nei PC moderni sfruttiamo uno standard di derivazione dal vecchio PCI, compatibile con esso, che è il *PCI Express* (e che non studieremo).

1.1.1 Indirizzamento dei dispositivi

Secondo lo standard PCI, separiamo il **bus locale** (quello che abbiamo visto finora) da un eventuale **albero di bus**, collegati fra di loro dai cosiddetti **ponti**. Il **ponte ospite-PCI**, in particolare, collega il **bus principale** (il più vicino al bus locale) al bus locale, mentre questo a sua volta viene collegato ad altri bus attraverso **ponti PCI-PCI**. Ad esempio, molti calcolatori dell'epoca erano dotati di *bus ISA* collegati con appositi ponti al bus principale, per la gestione di vecchie interfacce ISA.

A ogni bus è associato un numero su 8 bit, col bus principale che si prende il numero 0.

Per indirizzare un dispositivo usiamo invece 16 bit, disposti come:

Scopo	Bit	Max
Bus	8 bit	256
Device	5 bit	32
Function	3 bit	8

dove il numero di bus è lo stesso di prima.

Il numero di dispositivo differisce dal *Device ID*, che vedremo fra poco, e deriva dalla posizione fisica del dispositivo nel bus.

Il numero di funzione, invece, è reso necessario da schede che implementano più funzionalità, quali ad esempio le schede grafiche moderne, che si occupano anche dell'audio. In ogni caso, la funzione 0 deve essere implementata obbligatoriamente.

Come vediamo dalla tabella, poi, si possono avere direttamente dalle codifiche fino a 256 bus diversi, con 32 dispositivi ciascuno e 8 funzioni per dispositivo.

1.1.2 Operazioni coi dispositivi

Veniamo quindi a come funzionano le operazioni più semplici sul Bus PCI. Ogni operazione sul PCI viene detta **transazione**, ed ha un **iniziatore** e un **obiettivo**, cioè il dispositivo che inizia la transazione e il dispositivo che gli risponde.

L'iniziatore delle richieste che il nostro programma invia all'I/O, notiamo, non sarà più il processore, ma il ponte ospite-PCI. Inoltre, il bus PCI permette in verità che anche dispositivi esterni al ponte ospite-PCI facciano da iniziatori per transazioni, rendendo possibili meccanismi come il **DMA** (*Direct Memory Access*).

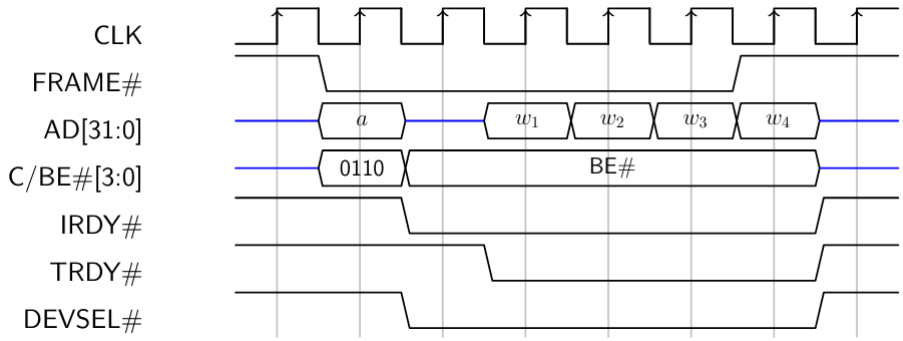
Sul bus vero e proprio troviamo quindi:

- La linea di **clock**, che tutte le interfacce vedono, originariamente intorno ai 33 MHz. Sul fronte di salita del clock tutte le interfacce (idealmente) campionano i segnali sul bus;
- **FRAME#**, che "incornicia" la transazione corrente: l'iniziatore lo alza quando la transazione è finita. Al massimo, a handshake avvenuto si possono avere un numero indefinito di cili di trasmissione da 4 byte: il FRAME si alza ad avvenuta trasmissione dell'ultimo fra questi;
- **AD**, la linea *condivisa* di indirizzo o dati, a 32 bit;
- **C/BE#**, *controllo* e *byte-enable*, codificano il tipo di operazione in fase di indirizzamento e fanno da byte-enable nel trasferimento dati;
- **IRDY#** e **TRDY#**, rispettivamente *Initiator Ready* e *Target Ready*, supportano l'handshake nella fase di scambio dati: l'iniziatore abbassa IRDY quando è pronto a ricevere dati o quando inizia a scriverli, mentre l'obiettivo abbassa TRDY quando inizia a inviare dati o a riceverli;
- **DEVSEL#**, viene attivato dal dispositivo che riconosce, controllando C/BE e l'indirizzo su AD, una chiamata a sé stesso, quindi dal presunto *obiettivo*;
- **STOP#**, viene attivato dall'*obiettivo* per terminare prematuramente una transazione.

dove il # indica *attivi bassi*.

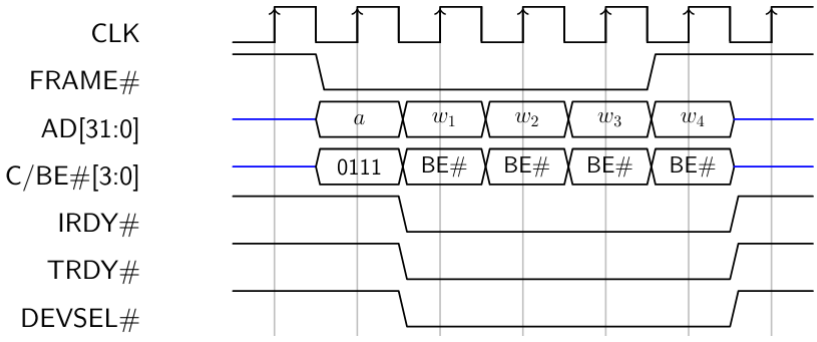
Vediamo nel dettaglio due transazioni complete, di lettura e scrittura a 4 byte sequenziali, per capire a pieno la politica di handshake:

- **Lettura:**



Abbiamo che l'iniziatore come prima cosa porta il FRAME in basso e scrive l'indirizzo in AD e la modalità di indirizzamento in C/BE (qui `0110`, che significa memoria). L'obiettivo risponde abbassando DEVSEL, al cui l'iniziatore risponde abbassando IRDY. Fra l'abbassamento di IRDY e l'inizio della trasmissione (l'abbassamento di TRDY) c'è un ciclo di clock a vuoto, detto *ciclo di turnaround*, che permette all'iniziatore di rilasciare AD e C/BE perchè l'obiettivo possa assumerne il controllo. Si trasmettono quindi 16 byte in mandate da 4, con FRAME che torna alto al quarto byte per segnalare la fine dell'operazione, e le linee di handshake (IRDY, TRDY e DEVSEL) tornano alte.

- **Scrittura:**



Le cose si svolgono in maniera pressoché identica, con la differenza che non si necessita di un ciclo di turnaround in quanto l'iniziatore sarà l'unico a scrivere su AD e C/BE. Inoltre, possiamo immaginare che C/BE venga effettivamente modificato nel corso della scrittura, in quanto magari non vogliamo impattare tutti i byte ad ogni ciclo di scrittura.

1.1.3 Spazio di configurazione

Per poter lavorare con il bus PCI, poi, introduciamo un nuovo spazio indirizzabile, quello di **configurazione**. Avremo quindi che il processore può indirizzare:

- **Memoria**;

- **I/O**;
- **Configurazione**.

Questo spazio è rilevante solo all'avvio del calcolatore (all'esecuzione del BIOS, o come definito originariamente dallo standard, *PCI BIOS*), appunto per effettuare la configurazione delle interfacce PCI installate nel sistema. Ogni bus PCI trasporta quindi messaggi in uno qualsiasi di questi 3 spazi, e si discrimina lo spazio specifico controllando cosa l'iniziatore mette in C/BE.

Useremo lo spazio di I/O come sempre, ma i segnali del processore viaggeranno attraverso i vari ponti fino all'interfaccia desiderata. Per quanto riguarda i dispositivi mappati in memoria (si pensi al video), invece, possiamo assumere che all'avvio il ponte ospite-PCI deve solo sapere la dimensione della memoria RAM installata, in modo da rispondere da lì in poi solo agli indirizzi *di memoria* posti al di *sotto* di essa. Avevamo visto questo meccanismo nella sezione 17.2, vedendo come il bootloader ignora la zona di memoria superiore alla RAM e dedicata al bus PCI.

Notiamo che tutto questo sistema è comunque strutturato per essere trasparente al processore, e quindi invisibile lato software.

1.1.4 Configurazione dei dispositivi

Ogni dispositivo sul bus è obbligato a fornire, per ogni funzione e ad una certa locazione predefinita, un numero di registri che formano 64 righe da 32 bit, che devono contenere informazioni di configurazione. I primi due dati, su 16 bit (quindi una riga), saranno il **Vendor ID** e il **Device ID**, seguiti da altri dati che non ci sono immediatamente rilevanti:

<table>
<tr><th>+3</th><th>+2</th><th>+1</th><th>+0</th><th>offset</th></tr>
<tr><td colspan="2">Device ID</td><td colspan="2">Vendor ID</td><td>0x00</td></tr>
<tr><td colspan="2">Status</td><td colspan="2">Command</td><td>0x04</td></tr>
<tr><td colspan="3">Class Code</td><td>Revision ID</td><td>0x08</td></tr>
<tr><td>...</td><td>Header Type</td><td>...</td><td>Cache Line Size</td><td>0x0c</td></tr>
<tr><td colspan="4">Base Address Register 0</td><td>0x10</td></tr>
<tr><td colspan="4">Base Address Register 1</td><td>0x14</td></tr>
<tr><td colspan="4">Base Address Register 2</td><td>0x18</td></tr>
<tr><td colspan="4">Base Address Register 3</td><td>0x1c</td></tr>
<tr><td colspan="4">Base Address Register 4</td><td>0x20</td></tr>
<tr><td colspan="4">Base Address Register 5</td><td>0x24</td></tr>
<tr><td colspan="4">...</td><td>0x28</td></tr>
<tr><td colspan="2">...</td><td colspan="2">...</td><td>0x2c</td></tr>
<tr><td colspan="4">...</td><td>0x30</td></tr>
<tr><td colspan="3">...</td><td>...</td><td>0x34</td></tr>
<tr><td colspan="4">...</td><td>0x38</td></tr>
<tr><td>...</td><td>...</td><td>Intr. Pin</td><td>Intr. Line</td><td>0x3c</td></tr>
</table>

Il Vendor ID è determinato da un'autorità centrale (la *PCI-SIG*): ad esempio, il vendor ID della Intel è `0x8086`.

Per permettere quindi alla CPU di configurare i dispositivi, cioè accedere ai loro registri di configurazione, il ponte ospite-PCI rende disponibili alla CPU due registri, entrambi su 32 bit:

- Il **CAP**, *Configuration Address Port*, che permette di selezionare una funzione e l'offset della parola a cui si vuole accedere;
- Il **CDP**, *Configuration Data Port*, che permette di accedere alla parola selezionata con *CAP*.

Le operazioni effettuate dalla CPU attraverso questi due registri verranno trasformate automaticamente dal ponte ospite-PCI in operazioni di configurazione sui bus PCI.

La posizione di questi registri in memoria è la seguente:

`0xcf8`	**CAP**, *Configuration Address Port*
`0xcfc`	**CDP**, *Configuration Data Port*

Vediamo quindi cosa deve fare il BIOS per la configurazione dei dispositivi PCI, cioè per collocarne nello spazio di I/O o in memoria eventuali registri o porzioni di memoria, rispettivamente. Ogni dispositivo ha una **dimensione naturale** che occupa nello spazio, sia questo di memoria o di I/O. Fornisce quindi al processore un registro, detto **BAR** (*Base Address Register*), che è scrivibile solo in parte: la parte meno significativa, infatti, è fissa a 0 e determina la dimensione naturale della regione che questo occuperà. Per rilevare la dimensione naturale, quindi, basta scrivere tutti 1 sul BAR e controllare quali bit vengono effettivamente modificati.

Il PCI BIOS dovrà quindi, attraverso i registri CAP e CDP, controllare il BAR di tutte le interfacce, determinarne la dimensione naturale e trovare una regione libera nello spazio di I/O o in memoria, a seconda del tipo di dispositivo, dove collocarle.

1.1.5 Interruzioni PCI

Per la gestione delle interruzioni, lo standard si ferma al dire che ogni dispositivo deve specificare per ogni funzione quale, di quattro linee di interruzione, tale funzione usa, in un apposito registro di configurazione. Queste linee sono dette **INTA**, **INTB**, **INTC** e **INTD**, e lo spazio di configurazione della funzione contiene il dato rispetto a quale linea usa in *Intr. Pin*, con 0 che significa nessuna interruzione, 1 *INTA*, e così via.

1.1.6 Struttura di un bus PCI

Per fissare i concetti di quest'ultima sezione, cerchiamo di esporre la struttura del calcolatore emulato su cui abbiamo studiato finora. Scriviamo quindi un programma che cerchi tutti i dispositivi presenti nella configurazione attuale, sfruttando le funzioni per la gestione dello spazio di configurazione definite in `libce.h` (`read_conf...()`):

```
1 #include <libce.h>
2
3 bool check_dev(natb bus, natb dev, natb fun) {
4   // entrate tabella configurazione
5   natw vendorID, deviceID;
6   natb class_code;
7
```

```
  // ottieni vendor ID
  vendorID = pci::read_confw(bus, dev, fun, 0);

  // 0xffff significa inesistente
  if(vendorID == 0xffff) return false;

  // ottieni device ID e codice classe
  deviceID = pci::read_confw(bus, dev, fun, 2);
  class_code = pci::read_confb(bus, dev, fun, 11);

  // stampa informazioni dispositivo
  printf("%02x:%02x.%1d   %04x:%04x [%s]\n",
    bus, dev, 0,
    vendorID, deviceID,
    pci::decode_class(class_code)); // decodifica il codice classe

  return true;
}

void main() {
  for(natb bus = 0; bus < 100; bus++) { // 100 bastano
    for(natb dev = 0; dev < 32; dev++) {
      // cerchiamo il dispostivo bus:dev:xxx
      if(!check_dev(bus, dev, 0)) continue;

      // controlla le altre funzioni
      for(natb fun = 1; fun < 8; fun++) {
        check_dev(bus, dev, fun);
      }
    }
  }

  pause();
}
```

e vediamo cosa stampa:

```
00:00.0   8086:1237 [bridge device]             % ponte ospite-PCI
00:01.0   8086:7000 [bridge device]             % ponte PCI-ISA
00:01.1   8086:7010 [mass storage controller]   % ponte PCI-ATA
00:01.3   8086:7113 [bridge device]             % altre funzioni PIIX3
00:02.0   1234:1111 [display controller]        % scheda VGA virtuale
```

dove i commenti sono stati aggunti successivamente.

Vediamo quindi che abbiamo 3 dispositivi, tutti sul bus 0 (bus principale):

0: Il ponte ospite-PCI;

1: Un dispositivo che fornisce altri bus, che possiamo individuare nel *southbridge* Intel **PIIX3** (*PCI IDE ISA Xcelerator*). Questi sono:

   0: Un bus ISA accessibile attraverso il ponte PCI-ISA, che emula le interfacce di tastiera e di timer dell'ISA;

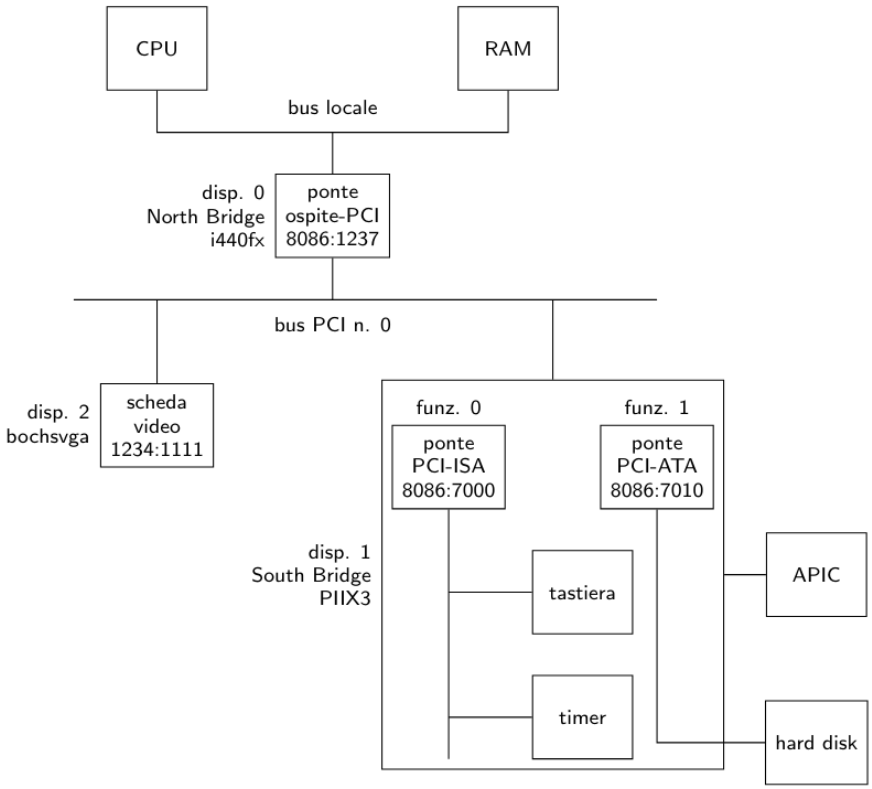
   2: Un bus ATA accessibile attraverso il ponte PCI-ATA, che permette l'interazione con il disco rigido (hard disk);

   3: L'interfaccia **ACPI** (*Advanced Configuration and Power Interface*), non di interesse a questo corso.

Il southbridge si interfaccia poi con altri componenti, quali ad esempio l'APIC (ed emula il PIC sul bus ISA).

2: La scheda video VGA emulata di *bochs*, un emulatore precedente a QEMU.

La struttura complessiva sarà quindi la seguente:



1.2 I/O nel kernel

La maggior parte delle modifiche allo spazio di I/O apportate introducendo il bus PCI, abbiamo detto, sono effettivamente trasparenti al processore, se non per l'introduzione dei registri CAP e CDP, che devono essere comunque usati una volta sola nel PCI BIOS per la configurazione dei dispositivi. Possiamo quindi sfruttare la maggior parte delle funzioni definite in `libce.h` per l'input/output senza particolari problemi.

Vediamo quindi com'è implementata la gestione dell'I/O da parte del *kernel*.

Avevamo detto che la motivazione principale dietro lo sviluppo del sistema multiprogrammato era che questa permetteva l'interruzione in qualsiasi momento di un *processo* in esecuzione, per permettere al processore di effettuare altre operazioni. Le interruzioni sollevate da i dispositivi esterni, vediamo, possono essere gli eventi che provocano tale cambio di contesto, e quindi la gestione del segnale in ingresso al sistema.

Ad esempio, un processo potrebbe, con la funzione `readconsole()`, specificare un buffer e un numero di caratteri che vuole leggere da tastiera. A questo punto, il sistema lo metterà in attesa e si occuperà di altro, riempiendo sequenzialmente il buffer via via che i tasti vengono effettivamente premuti (e quindi le relative interruzioni sollevate). Una volta che il buffer sia riempito dal numero di caratteri richiesti dal processo, potrà quindi rimettere il processo in esecuzione, o nella lista pronti, e proseguire.

Per tenere conto di più processi che possono voler leggere contemporaneamente, poi, ci dotiamo di un semaforo che tenga conto di chi sta usando quella risorsa in un certo momento. Un problema qui sarà come usare i semafori da lato sistema, e verrà discusso nella prossima lezione.