Appunti Calcolatori Elettronici

Luca Seggiani
2025

1 Lezione del 24-02-25

1.1 Introduzione al corso

Continuiamo lo studio di una particolare architettura per calcolatori, a partire da quanto detto riguardo alle reti logiche, introducendo i concetti di **interruzione**, **protezione** e **memoria virtuale**. Questi 3 strumenti ci permetterano di realizzare il paradigma della **multiprogrammazione**, cioè di far eseguire ad una macchina con un singolo processore più programmi contemporaneamente. Non si pensi questo significhi avere più processori, in quanto il corso riguarda esclusivamente processori *single-threading*.

1.2 Architettura

L'architettura di riferimento è quella classica, composta da **CPU**, **memoria** e **I/O** interconnessi da un **bus**.

Durante lo studio di un archiettura è oppurtuno porsi la domanda *"chi fa cosa?"*, che fornisce determinati *chi* ai determinati *cosa* forniti da un opportuno livello di astrazione (transistor, porte logiche, diagrammi funzionali, ecc...).

La domanda che potremo porci adesso è *"chi comanda?"* all'interno dell'architettura vista. La risposta più giusta è quella del **software**: l'architettura è fatta per *eseguire* software.

Per convincerci di questo possiamo sostituire la domanda *"chi fa cosa?"* con la domanda *"chi sa cosa?"*.

- La **CPU** conosce lo stato corrente dei registri e l'istruzione in esecuzione. Fra un'istruzione e l'altra non c'è alcun bisogno di sapere cosa è accaduto finora, e cosa accadrà in futuro, ma solamente l'istruzione corrente. Quindi si può pensare che la CPU non *sa* qual'è l'obiettivo della computazione, ma si limita a portarla avanti.

- La **memoria** è un oggetto passivo, che contiene il programma, ma si limita a restiture i dati richiesti quando sono richiesti. Notiamo che le memorie che usiamo sono ad **accesso casuale**, ergo nessuno scorre alla ricerca di indirizzi, ma si può leggere e scrivere in posizioni arbitrarie in tempo pressoché costante. La memoria contiene **sempre** qualcosa, che questo sia significativo o meno, e la sua tipizzazione dipende solamente dalle intenzioni del programmatore.

- L'**I/O** è il componente più variegato dell'architettura. L'unica costante che rende la comunicazione con le periferiche più facile è la presenza di un interfaccia, che riduce tale comunicazione ad una semplice lettura o scrittura nello spazio di I/O. La differenza fra le letture e scritture nello spazio di I/O e lo spazio di memoria è la possibile presenza di **effetti collaterali**, cioè effetti non riconducibili alla sola variazione di stato di una locazione di memoria. Inoltre la CPU non è l'unica a scrivere nello spazio di I/O, in quanto questo può essere fatto anche dalle periferiche stesse.

- Il **bus** è un insieme di linee (*fili*), che trasportano ciò che ogni componente sta comunicando in un dato momento. Ogni componente vede ciò che viene scritto sul bus in qualsiasi momento, e l'indirizzamento di locazioni specifiche nello spazio di memoria o nello spazio di I/O viene fatto attraverso **maschere** di indirizzo.

1.2.1 Flusso di controllo

Abbiamo visto come la CPU si limita a prelevare ed eseguire istruzioni nel ciclo di **fetch-execute**. L'istruzione successiva alla corrente, il cui indirizzo viene scritto nell'**instruction pointer**, viene decisa dall'istruzione corrente stessa (si pensi alle istruzioni di salto). Il **flusso di controllo** è quindi deciso dall istruzioni stesse, cioè dal programma.

1.2.2 Bootstrap

Il **bootstrap** è un processo secondo il quale si porta il sistema in un certo stato di esecuzione, apparentemente impossibile, o comunque molto difficile, da raggiungere. Ad esempio, il compilatore del linguaggio C è scritto esso stesso in linguaggio C. La domanda naturale è *"come è stato compilato il compilatore?"*. La risposta è un processo di boostrap, usando o un compilatore presesistente, magari che implementa un sottoinsieme parziale del C, o scrivendo l'intero compilatore in linguaggio macchina, cioè assemblando codice assembler.

Il boostrap si rende necessario anche all'avvio del calcolatore, per il caricamento del programma all'interno della memoria e l'inizio dell'esecuzione. Nei calcolatori moderni questo viene fatto attraverso la **ROM**, cioè una memoria a sola lettura che contiene un programma di bootstrap. All'avvio il processore è impostato in modo che al reset prenda come indirizzo proprio quello della ROM, e quindi inizi ad eseguire il programma di boostrap. All'interno della ROM si trova, nei calcolatori moderni, il **BIOS** (o *UEFI*, nei sistemi moderni), che ha il solo compito di impostare alcune periferiche di base e caricare il sistema operativo.

Iniziamo quindi ad approfondire, uno per uno, i moduli dell'architettura.

1.3 Memoria

La memoria è un insieme contiguo di locazioni di memoria, che nelle architetture moderne sono rappresentate da byte. Storicamente, la memoria era indirizzata a *parole*, cioè insiemi di bit coincidenti in dimensioni coi registri del processore. Una parola poteva essere di più byte, mentre oggi le memorie sono accessibili ai singoli byte. Ad esempio, le memorie usate nell'architettura Intel x86 sono accessibili ad 1 byte (`MOVB`), 2 byte (`MOVW`), 4 byte (`MOVL`), e 8 byte (`MOVQ`).

1.3.1 Endianess

Notiamo che la posizione in memoria del byte più significativo di una parola (in questo caso consideriamo una "parola" da 8 byte, da cui si ricavano tutte le altre misure) determina l'*endianess* dell'architettura. In particolare, se l'ultimo byte sta in fondo nella memoria, si dice **big-endian**, mentre se viceversa l'ultimo byte viene per primo nella memoria, si dice **little-endian**.

L'architettura Intel x86 che andiamo a considerare è little-endian, come lo sono la maggior parte delle architetture moderne. Un esempio di utilizzo del big-endian e nella trasmissione di dati attraverso il protocollo IP, usato nelle comunicazioni Internet.

1.3.2 Allineamento

Indicheremo con **offset** la distanza in byte fra due locazioni di memoria, inteso come il numero di locazioni che vanno saltate per raggiungere un indirizzo a partire dall'altro. In questo ha senso parlare anche di offset *negativi*.

Visto che lo spazio di memoria è effettivamente ciclico, cioò si ha *wrap-around* ai suoi capi, si ha che gli offset rimangono validi **modulo** la dimensione dello spazio di memoria, che è sempre 2^n, con n nel nostro caso uguale a 64.

Il *wrap-around* si comporta bene con gli offset, ma lo stesso non si può dire per quanto riguarda **intervalli** di byte. Preso un certo intervallo $[x, y)$, quindi, si ha che questo contiene gli indirizzi $\{n \mid x \leq n < y\}$, ammesso che $x < y$, cosa che risulta falsa nel caso di intervalli che hanno *wrap-around*. Decidiamo di non considerare intervalli di questo tipo. Questo rende necessaria un'eccezione per intervalli che comprendono l'ultimo byte: in questo caso è concesso $[x, 0)$, con 0 che indica il fondo dello spazio di memoria.

Veniamo quindi all'**allineamento**. Dire che un indirizzo è allineato ad un numero n significa dire che quell'indirizzo è un multiplo di n. Chiaramente, conviene scegliere n potenze di 2. In questo caso, per riconoscere se un indirizzo è allineato a 2^k, basta guardare i suoi primi k bit.

Si dice spesso che oggetti sono *allineati alla parola*, ecc... Questo significa che sono allineati alla *dimensione* della parola specificata. Altrimenti, si può dire che un oggetto è allineato *naturalmente*, nel caso in cui sia allineato alla dimensione di stesso.

Infine, il **confine** di un oggetto è l'indirizzo che lo delimita dal resto dello spazio di memoria.

2 Lezione del 25-02-25

2.1 Interazione fra CPU e memoria

Nell'architettura Intel x86 la CPU interroga la RAM in due situazioni:

- Durante la lettura di un istruzione;
- Durante la lettura di *eventuali* operandi in memoria richiesti dall'istruzione. Notiamo che per ogni istruzione è previsto un solo indirizzo esplicito di un operando in memoria (non è permesso scrivere qualcosa come `MOV (%RBP), (%RDI)`). indirizzo Alcune istruzioni possono però avere comunque più di un operando in memoria (ad esempio le istruzioni di stringa, `MOVS`, ecc... o la stessa istruzione di pila `POP`).

Dal punto di vista pratico, il collegamento fra CPU e RAM è rappresentato da:

- Un **bus dati** a 64 bit;
- Un certo numero di linee per il **numero di riga**. Questo non corrisponde all'indirizzo del primo byte contenuto in ogni riga, ma l'indice proprio di ogni regione (intesa come riga) da 64 bit all'interno della RAM. Si noti inoltre che queste non sono necessariamente 2^{64}, o 2^{57} (il massimo spazio indirizzabile secondo l'architettura x86), ma più spesso intorno alle 2^{36}-2^{37};
- Determinate **linee di controllo** che segnalano l'operazione in corso da parte del processore.
- 8 linee di **byte enable**, attive basse, che rappresentano i byte di interesse all'interno di ogni locazione da 64 bit della RAM. Dal punto di vista della lettura, queste linee non sono particolarmente utili in quanto tutta la locazione verrà comunque riportata sul bus dati, o comunque le locazioni non selezionate potranno essere invalide o in alta impedenza, senza avere effetto sulla CPU (che non le leggerà). Per quanto riguarda la scrittura, invece, la RAM lascerà inalterati i byte con byte enable alto.

2.1.1 Struttura della RAM

Modelliziamo un modulo di RAM come una rete provvista di:

- Una linea di **select**, attiva bassa;
- Le **linee di indirizzo**;
- Una linea di *memory read* e una linea di *memory write*, o comunque un certo numero di **linee di controllo** necessarie all'accesso in scrittura e lettura;
- Un **bus dati** di ingresso/uscita.

Dalla CPU arriveranno, come abbiamo detto, i **numeri di riga**, i **byte enable**, il **bus dati** e le **linee di controllo**.

I numeri di riga si collegano direttamente alle linee di indirizzo di ogni modulo, che rappresenterà un certo byte della locazione (avremo quindi, nell'architettura descritta, 8 moduli per 8 byte, quindi 64 bit). I byte enable dovranno quindi smistarsi nelle linee di select di ogni modulo di RAM, a selezionare il modulo corrispondente. Il bus dati verrà composto, analogamente, concatenando le linee di uscita da 8 bit di ogni modulo di RAM. Notiamo che avevamo chiamato questo montaggio **parallelo**.

Vorremo poter estendere la memoria disponibile oltre il numero di locazioni da un byte fornite da ogni modulo di RAM. Pensiamo di fare questo attraverso più banchi di memoria con locazioni da 64 bit. In questo caso avremo bisogno di montaggio in **serie**, e quindi di generare un segnale di select a partire non solo dalle line di byte enable, ma anche da una **maschera** generata a partire dal numero di riga. Questo si potrà fare agevolemnte mettendo il segnale di uscita della maschera in OR (ricordiamo segnali attivi bassi, quindi si applica De Morgan) con il byte enable di ogni modulo di RAM compreso nel banco di memoria associato a tale maschera.

2.1.2 Allineamento e RAM

Quanto discusso finora rende più chiaro l'importanza del corretto allineamento degli oggetti in memoria. Leggere un oggetto da 8 byte non allineato nel montaggio di RAM descritto, infatti, richiederà necessariamente 2 accessi, contro il singolo accesso necessario per un oggetto allineato. Inoltre, alcuni dei byte più significativi risulteranno invertiti di posto rispetto ai byte meno significativi, cioè si richiede un operazione di shift interna al processore.

Questa combinazione di operazioni, eseguite in **hardware**, rende gli accessi in memoria non allineati molto poco performanti, e quindi sconsigliati (anche se l'architettura Intel x86 li permette comunque).

Un problema che potrebbe interessarci è, data una regione di memoria $[x, y)]$ di dimensione b uguale a un singolo banco di RAM, ottenere gli indici della prima regione in cui cade l'intervallo, e la prima in cui non cade più.

Vediamo come calcolare la prima regione di appartenenza. In **hardware**, questo può essere calcolato semplicemente prendendo gli $n - b$ bit più significativi dei numeri di riga x e y.

In **software**, questo equivarrà ad uno shift a destra che conservi i soli $n - b$ bit più significativi.

Vediamo come calcolare l'offset di x o y all'interno delle rispettive regioni. Mascherando gli stessi bit, invece, si può ottenere l'indirizzo all'interno del banco del confine

della regione. Per la precisione, vogliamo una maschera fatta da $n - b$ 0 e b 1. Questa si può ricavare agevolmente prendendo 2^b come `1UL << b` e sottrandogli 1, ottenendo la maschera desiderata (si avranno borrow propagati dal bit in b fino al LSB).

Infine, vediamo come calcolare la prima regione di non appartenenza. In questo caso potremo calcolare la regione in cui cade $y-1$, e aggiungervi 1 (tenendo conto di eventuali *wrap-around*). Il -1 è richiesto dal fatto che y potrebbe cadere sul confine. In questo caso avremo `((y - 1) >> b) + 1`, considerata somma modulo $n - b$. Alternativamente, si può prendere $y + b$ e calcolarne la regione di appartenenza.

2.2 Spazio di I/O

Veniamo quindi alla trattazione dello spazio di I/O e delle interfacce ivi connesse. L'accesso alle periferiche viene fatto attraverso le istruzioni `IN` e `OUT`, ammesso che non ci sia nessun sistema operativo in esecuzione, ma solo il nostro programma, e appositi sottoprogrammi di ingresso/uscita, la cui struttura non è al momento importante.

Le periferiche che studieremo, per semplicità di trattazione, derivano in parte da quelle disponibili sui PC **IBM AT** (famiglia *IBM 5170*). I PC di questa categoria (compresi tutti i vari *IBM compatible*) si basavano sullo standard per periferiche **ISA** (*Industry Standard Architecture*). Visto che i PC moderni derivano dai vecchi IBM compatible, anche oggi si cerca di emulare (almeno in parte) questo standard.

Le periferiche, nello specifico saranno:

- La **tastiera**;
- Il **video** su VGA;
- Il **timer**;
- Gli **hard disk**.

2.3 Tastiera

Dal punto di vista funzionale, la tastiera deve solo scoprire quali tasti sono premuti e comunicarlo al calcolatore. In particolare, noi studieremo tastiere IBM che trasmettono secondo lo standard PS/2.

Nei PC IBM il tasto non restituisce il carattere ASCII del carattere premuto, ma un codice associato ad ogni tasto che va convertito in software. Questo codice viene ottenuto per *scansione* dell'intero piano della tastiera. Dal punto di vista meccanico, ci sono **tracce** orizzontali e verticali disposte, rispettivamente, su ogni riga o colonna di tasti. La pressione di un tasto comporta una deformazione delle tracce che chiude un circuito fra la riga e la colonna del tasto corrispondente. Un **microcontrollore** (originariamente un Intel 8042) collegato sia alle tracce orizzontali che alle tracce verticali scansiona ciclicamente, con impulsi, o le righe leggendo le colonne, o le colonne leggendo rige, cercando un circuito chiuso. Un cortocircuito viene quindi rilevato dal microcontrollore, che aggiorna una (piccola) memoria interna con il tasto premuto. Di conseguenza, invia al calcolatore un segnale che codifica quali tasti sono stati premuti rispetto al precedente istante temporale, e quali tasti sono stati rilasciati rispetto al precedente istante temporale.

La tastiera non restituisce solo pressioni di tasti, ma anche i loro rilasci, cosa che può essere utile per ottenere combinazioni di tasti, pressioni estese nel tempo, ecc... I codici di pressione si dicono **make code**, mentre i codici di rilascio si dicono **break code** La

stessa pressione ripetuta di un tasto quando l'utente lo tiene premuto per un certo istante temporale era, nei PC IBM, realizzata direttamente nella tastiera (tecnologia *type-matic*), tra l'altro con periodo configurabile. Tramite il *type-matic*, su appositi tasti abilitati, si ha infatti una ripetizione dell'evento di *pressione* (non rilascio) di un tasto a frequenza costante dopo un intervallo di pressione continua.

Lato calcolatore, il segnale prodotto dal microcontrollore della tastiera viene letto da un interfaccia provvista dei seguenti registri:

- **RBR**, *Receive Buffer Register*;
- **TBR**, *Transmit Buffer Register*;
- **STR**, *Status Register*;
- **CMR**, *Command register*

RBR e TBR, come STR e CMR, condividono gli indirizzi, rispettivamente `0x60` e `0x64`. Il RBR conterrà i make e break code, mentre l'STR conterrà i flag di stato sia per RBR che per TBR (rispettivamente ai bit 0 e 1).

Potremmo chiederci il significato di un registro di trasmissione TBR. Questo serve, ad esempio, a governare i led di stato per funzioni speciali quali Caps-Lock, Num-Lock, Scroll-Lock ecc... nonché a modificare le impostazioni del type-matic e, in maniera completamente slegata alla tastiera, a provocare il reset del PC, scrivendo `0xFE` in CMR.

Vediamo quindi un programma C++ per l'interazione con l'interfaccia di tastiera. Notiamo che la libreria all'header `libce.h` definisce alcuni tipi (qui `natb`, un naturale su 8 bit, e `ioaddr`, un indirizzo nello spazio di I/O) e funzioni (qui `inputb`, ottieni byte dallo spazio di I/O, e `vi::char_write()`, stampa un carattere a schermo).

```
#include <libce.h>
#define NUM_CODES 28

// indirizzi porte tastiera
const ioaddr rbr_addr = 0x60;
const ioaddr str_addr = 0x64;

// tabella make code
natb make_codes[] = {
  0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, 0x18, 0x19,
  0x1e, 0x1f, 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26,
  0x2c, 0xd, 0x2e, 0x2f, 0x30, 0x31, 0x32,
  0x1c, 0x39
};

// tabella caratteri minuscoli
char l_table[] = {
  'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p',
  'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l',
  'z', 'x', 'c', 'v', 'b', 'n', 'm',
  '\n', ' '
};

// tabella caratteri maiuscoli
char u_table[] = {
  'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P',
  'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L',
  'Z', 'X', 'C', 'V', 'B', 'N', 'M',
  '\n', ' '
```

```
};

// make code ESC
natb esc_code = 0x01;

// make code shfit
natb shift_down = 0x2A;
natb shift_up = 0xAA;

// gestione case
enum cas { lower, upper };
cas cur_cas = lower;

natb get_key() {
  natb status;

  // ciclo di lettura per il flag FI in str_addr
  do {
    status = inputb(str_addr);
  } while(!(status & 0x01));

  // FI alto, leggi da rbr_addr
  return inputb(rbr_addr);
}

char get_char(natb make_code) {
  // cerca il carattere per scansione lineare
  for(int i = 0; i < NUM_CODES; i++) {
    if(make_code == make_codes[i]) {
      // trovato, controlla il case corrente
      if(cur_cas == upper) {
        return u_table[i];
      } else {
        return l_table[i];
      }
    }
  }

  // carattere nullo come default
  return '\0';
}

void main() {
  while(true) {
    // otteni make code
    natb make_code = get_key();

    // se ESC, esci
    if(make_code == esc_code) {
      break;
    }

    // gestisci shift
    if(make_code == shift_down) {
      cur_cas = upper;
    }
    if(make_code == shift_up) {
      cur_cas = lower;
    }

```

```
90      // ottieni carattere e stampa
91      char c = get_char(make_code);
92      vid::char_write(c);
93    }
94  }
```

Dal programma si evincono subito gli indirizzi dei registri di Receive Buffer (RBR) e di stato (STR), a cui i registri trasmettitore e comando (TBR e CMR) sono sovrapposti. Il funzionamento è quindi ottenuto attraverso una lettura ciclica dei make code dall'interfaccia, e una scansione per il rilevamento del carattere selezionato a partire dal make code stesso. Altro codice è usato per gestire il tasto shift, e il termine dell'esecuzione alla pressione del tasto ESC.

3 Lezione del 03-03-25

3.1 Video

Il supporto principale al video è la **memoria video**, che lato software si comporta perlopiù come una normale memoria ad accesso casuale.

Questo è quindi il primo esempio di un oggetto che si trova nello spazio di memoria, senza necessariamente *essere* memoria: ciò che vi viene scritto non viene memorizzato, ma visualizzato sullo schermo.

Inoltre, la memoria video supporta un accesso *bidirezionale*: cioè vi si può accedere sia lato CPU che lato **adattatore video**, cioè la rete che si occupa di gestire tale memoria e visualizzarla sul *display*. Lo standard VGA usato dal PC IBM prevede che l'adattatore sia configurabile e utilizzabile in due modalità:

- **Modalità testo:** ogni locazione viene associata ad un carattere ASCII da visualizzare sullo schermo, diviso in 80 colonne $\times$ 25 righe (con un carattere di 9 $\times$ 16 pixel, per una risoluzione totale di 720 $\times$ 400 pixel a 70 Hz). È questa la modalità di default in cui si avvia l'adattatore.

  In questo caso il compito dell'adattatore è quello di leggere i 2 KB di memoria, e convertire ogni codice nel carattere principale. Questo viene fatto consultando una ROM di caratteri che contiene quello che è effettivamente il carattere (*font*) dell'adattatore. Solitamente si può anche redirezionare la lettura in ROM ad una certa regione della RAM, modificando così il font.

  La faccenda è veramente più complicata: si dedicano non 1 ma 2 byte ad ogni carattere, dove il byte più significativo rappresenta informazioni riguardo al **colore** del carattere:

  - I 4 bit meno significativi rappresentano il colore del *foreground*;
  - I 3 bit successivi rappresentano il colore del *background*;
  - Il bit più significativo rappresenta il *blinking*, cioè indica all'adattatore di far *lampeggiare* quel carattere nel tempo.

  La modalità testo non ha idea della posizione del cursore sullo scherm: attraverso registri si può indicare la posizione del cursore, e modificando la regione di memoria interessata si possono cambiare i caratteri in qualsiasi zona dello schermo. Il comportamento del cursore (spostamento, ritorno a capo, ritorno carrello, ecc...) è quindi gestito interamente lato software.

- **Modalità grafica:** programmando i registri dell'adattatore si possono ottenere diverse modalità grafiche, che permettono al progammatore di colorare singoli pixel sul display. Le modalità più popolari di questo tipo su schede VGA erano o a 640 $\times$ 400 pixel a 70 Hz (fase di boot grafica), o a 640 $\times$ 480 pixel a 60 Hz non interlacciati (la modalità di default di Microsoft Windows). Nella macchina virtuale usata incapsuliamo tale operazione di conversione in un apposita libreria, e scriviamo pixel con colori su 8 bit (per 256 colori diversi). Nei sistemi moderni la memoria video non viene scritta dalla CPU, ma da un *coprocessore grafico* che esegue un suo programma, mentre la CPU può dedicarsi ad altro.

3.1.1 Indirizzamento dei registri dell'adattatore video

Vediamo nel dettaglio come si possono indirizzare i registri interni dell'adattatore video. Questo dispone infatti di una vasta gamma di registri, ma una sola linea di ingresso da un byte per indirizzamento e scrittura. Le scritture vengono quindi eseguite inserie:

- Prima specificando l'**indirizzo** del registro da aggiornare;
- Poi inserendo i **dati** da scrivere a tale indirizzo.

inpulso

Vediamo quindi un programma di esempio che sfrutta l'adattatore grafico in modalita testo a 25 $\times$ 80 caratteri. Dalla libreria `libce` si è importata la variabile `video`, che rappresnta l'intero buffer da 2 KB di memoria video (2 byte per cella) a disposizione dall'adattotore. Le scritture vengono fatte, come avevamo detto, sovrascrivendo dati in tale buffer, mantenendo *lato software* un cursore che ci indica la posizione sullo schermo.

```
#include <libce.h>
// dimensioni memoria video (2K)
#define SIZE 2000

namespace vid {
  // dichiara l'array video di libce
  extern volatile natw* video;
}

#define video vid::video

char mess[] = "- x86 rules ";
int cursor = 0;

// attributi carattere: 0x00001111-ASCII-
// significa bianco su sfondo nero
natl attr = 0x0F00;

// stampa una stringa
void prt_string(char* mess) {
  while(*mess != '\0') {
    video[cursor] = *mess | attr;
    cursor = (cursor + 1) % SIZE;
    mess++;
  }
}

void main() {
  for(int i = 0; i < 167; i++) {
    prt_string(mess);
```

```
  }

  do {
    // esci su ESC
    natb k = kbd::get_code();
    if(k == 0x01) break;
  } while(true);
}
```

3.2 Timer

Il timer è realizzato come un interfaccia ad eventi, che riceve in ingresso un clock e aggiorna ciclicamente un registro contatore. Al raggiungimento di 0 da parte del contatore, si resetta e si invia un certo evento (un impulso).

Nel PC IBM in particolare troviamo 3 contatori:

- **Contatore 0:** è collegata al controllore delle interruzioni;
- **Contatore 1:** era storicamente usato per il refreseh della RAM, oggi non viene più usato;
- **Contatore 2:** era collegato all'unico dispositivo audio presente sull'IBM, cioè il beeper speaker.

Il timer dispone di una vasta gamma di parametri e di solo 2 piedini di indirizzo, quindi 4 locazioni nello spazio di I/O, che sono **CWR**, un registro comune di configurazione delle interfacce, più 4 registri per timer collegati ad ognuna delle 3 porte rimanenti. L'accesso ai registri di un singolo timer va quindi fatto, cosa interessante del chip, in serie, modificando la stessa porta 1 o 2 volte consecutivamente.

3.2.1 Sonoro

Vediamo in particolare il lato sonoro del PC IBM. Essendo stato questo un calcolatore pensato per l'uso da ufficio, le capabilità audio erano molto limitata: si disponeva di un beeper speeker a frequenza modulabile dal timer (contatore 2). Inoltre, un particolare registro in memoria era collegato direttamente in AND con l'uscita del contatore 2, permettendo la modulazione on/off del segnale allo speaker. La modulazione in volume del pc speaker, in particolare, viene fatta attraverso un ulteriore registro detto **SPR**.

Questo tipo di modulazione permetteva effettivamente di sfruttare, in maniera non prevista dalla IBM, per riprodurre segnali generici.

Mostriamo un esempio di un programma per la riproduzione di un brano musicale monofonico, inteso come una sequenza di note:

```
#include "song.h" // include libce
#define TIMESTEP 3

// indirizzi timer
const ioaddr timer0_addr = 0x40;
const ioaddr timer2_addr = 0x42;
const ioaddr cwr_addr = 0x43;
const ioaddr spr_addr = 0x61;

// frame corrente di esecuzione
natl frame = 0;

```

```
// imposta il divisore di un timer
void set_divisor(natw divisor, ioaddr addr) {
  // metti divisor in addr in 2 passate
  outputb(divisor, addr);
  outputb(divisor >> 8, addr);
}

// leggi il conteggio di un timer
natl read_timer(ioaddr addr) {
  // comando di latch
  outputb(0x00, cwr_addr);

  natb low = inputb(addr);
  natb high = inputb(addr);
  return (high << 8) | low;
}

// abilita lo speaker
void note_on() {
  outputb(3, spr_addr);
}

// disattiva lo speaker
void note_off() {
  outputb(0, spr_addr);
}

bool update_song() {
  song_frame cur_frame = song[frame];

  switch(cur_frame.mode) {
    case 0:
      // off
      note_off();
      break;
    case 1: {
      // on
      note_on();

      // divisore della nota
      natl divisor = cur_frame.get_divisor();
      set_divisor(divisor, timer2_addr);
      break;
    }
    case 2: {
      // legato
      natl divisor = cur_frame.get_divisor();
      set_divisor(divisor, timer2_addr);

      break;
    }
  }

  frame++;

  if(frame == length + 1) return false;
  return true;
}

void main() {
```

```
  // imposta il timer 0
  // outputb(0x36, cwr_addr); // modo 3
  natl div = 0; // 0 significa 65536
  set_divisor(div, timer0_addr);

  // imposta il timer 2
  outputb(0xB6, cwr_addr); // modo 3

  // i tick svolti
  natl tick = 0;
  natl next_song_update = 0;

  natl last_value = read_timer(timer0_addr);

  while (true) {
    natl current_value = read_timer(timer0_addr);

    // se il valore corrente    maggiore del valore precedente
    // si    fatto un salto
    if (current_value > last_value) {
      tick++;
    }

    // aggiorna se necessario
    if(tick > next_song_update) {
      bool res = update_song();
      if(!res) break;

      // imposta il prossimo aggiornamento
      next_song_update += TIMESTEP;
    }

    last_value = current_value;
  }
}
```

In particolare, il programma si basa sullo scorrimento di un array di strutture `song_frame` (definite in `song.h`, la cui struttura non ci interessa) ad intervalli regolari. Questi intervalli vengono ottenuti configurando il timer 0 per oscillare in modalità 2 (oscillazione a onda quadra) con un divisore di 65536, che per un clock a 1.19 MHz fa:

$$\frac{1.19\,\text{MHz}}{65536} \approx 17.158\,\text{Hz}$$

Lato software, si aspettano 3 di queste oscillazioni, per ottenere una frequenza di $\approx$ 6Hz (abbastanza vicina a quella delle crome a 120 BPM).

Una volta ottenuto il frame, quindi, si aggiorna il timer 2 di conseguenza, sfruttando il registro SPR per il volume della nota (qui solo on/off) e il suo registro di scrittura per la nota stessa.

4 Lezione del 04-03-25

4.1 Hard disk

Gli **hard disk** (*dischi rigidi*) sono effettivamente, seppur memorie, **periferiche**, collegate al bus attraverso la loro interfaccia. La CPU non puo' eseguire programmi direttamente dall'hard disk, ma deve prima caricarli in memoria principale (memoria RAM).

Questo perchè letture e scritture in hard disk vengono effettuate per **blocchi** (storicamente di 512 byte), e richiedono molto più tempo di quanto sia possibile aspettare al prelievo di istruzioni o operandi.

Dal punto di vista elettromeccanico venivano realizzati attraverso dischi di materiale ferromagnetico imperniati ad un asse centrale, con testine mobili che scandivano il raggio dei dischi, rilevando o modificando la loro magnetizzazione per accedere all'informazione. Il complesso di dischi e testine viene detto **drive**.

L'informazione viene disposta su ogni disco in **settori** e **tracce**. Le tracce sono concentriche e i settori formano degli "spicchi" di ogni faccia. Notiamo che entrambe le facce di ogni disco possono memorizzare informazione. Un **blocco** è quindi formato dalla regione di una traccia compresa in un certo sensore.

I dischi vengono tenuti continuamente in rotazione (negli ordini delle centinaia/migliaia di RPM). Il tempo che la testina impiega a raggiungere una tracca viene detto **tempo di seek**, t_{seek}, il tempo che alla velocità di rotazione del disco l'informazione si trovi sotto la testina **latenza** $t_{latency}$ e il tempo necessario ad effettuare l'operazione vera e propria **tempo di lettura/scrittura** $t_{r/w}$, per cui il tempo di lettura/scrittura complessivo risulta:

$$t_{seek} + t_{latency} + t_{r/w} \sim 1\,\text{ms}$$

nell'ordine del millisecondo, per la CPU estremamente (milioni di volte) più lento della RAM.

Quello che accade al tempo di lettura è che il blocco viene copiato in un buffer di memoria nell'interfaccia che viene poi reso disponibile alla CPU. Viceversa, al tempo di scrittura il buffer viene riempito dalla CPU, e l'interfaccia si occupa poi di copiarlo all'interno del settore giusto.

Per effettuare un operazione dobbiamo quindi sapere:

- Quale *testina* individuare;
- Quale *traccia* individuare;
- Quale *regione* (quindi quale *blocco*) individuare.

Storicamente queste informazioni erano gestite lato software, concedendo la possibilità di alterare la *formattazione* del disco. Oggi la formattazione è definita in fabbrica, e l'interfaccia offre una sua astrazione. In questa astrazione ogni blocco è quindi indirizzato da un indirizzo logico, il **Logical Block Address**, **LBA**.

4.1.1 Interfaccia ATA

Nello standard PC AT gli hard disk usano interfacce **ATA** (capaci di gestire 2 drive, in configurazione *master/slave*). L'interfaccia ATA è dotata di diversi registri a 8 bit e uno a 16 bit:

- **Registri di selezione** del blocco:
  - **SNR** (Sector Number);
  - **CNL** (Cylinder Number Low);
  - **CNH** (Cylinder Number High);

  - **HND** (Head And Drive): solo gli ultimi 4 bit di questo registro formano l'informazione sulla testina da utilizzare. Gli altri bit vengono usati diversamente, ad esempio per selezionare quale drive usare in configurazioni master/slave, o per abilitare il LBA, usando quindi i registri di selezione per specificare un indirizzo logico (su $3 \cdot 8 = 4 = 28$ bit) anzichè un informazione geometrica sulla posizione del blocco desiderato.

  Vediamo che dalla dimensione dell'LBA (assumiamo che per indirizzamento geometrico si trova la stessa cosa) si ha una dimensione del disco:

$$2^{28} \cdot 2^9 = 2^{37} = 128\,\text{GB}$$

  Per questo si puo' abilitare la modalità **LBA48** (che non è un gruppo di idol giapponesi), dove ci si aspetta il LBA venga specificato in due passate, una da 24 bit e una da 20 bit sugli stessi registri.

- **SCR** (Section Counter): permette di specificare su quanti settori contigui a partire da quello specificato prima eseguire l'operazione;
- **BR** (Buffer Register): l'unico registro a 16 bit, permette di accedere al buffer 2 byte alla volta;
- **STS** (Status Register): il classico registro di stato che ci notifica se un'operazione è conclusa o si puo' effettuare;
- **CMD** (Command): serve a specificare l'operazione da effettuare (lettura, scrittura, ecc...).

Vediamo quindi un ultimo programma di esempio delle periferiche, che permette di scrivere un buffer di caratteri da 512 byte, stampandolo a schermo, e scriverlo/leggerlo su un settore di memoria ad un indirizzo LBA (prendiamo 1).

```
#include <libce.h>
#include "keyboard.h"
#include "video.h" // definisce il buffer video
#define BUF_SIZE 512

// registri disco
const ioaddr disk_buffer = 0x01F0;
const ioaddr disk_status = 0x01F7;
const ioaddr disk_sectors = 0x01F2;
const ioaddr disk_command = 0x01F7;

// registri indirizzo LBA (sarebbero SNR CNL CNH HND)
const ioaddr disk_lba0 = 0x01F3;
const ioaddr disk_lba1 = 0x01F4;
const ioaddr disk_lba2 = 0x01F5;
const ioaddr disk_lba3 = 0x01F6;

// dai indirizzo LBA al controllore disco
void give_lba(natl lba) {
  // dividi in 4 byte
  natb lba0 = lba;
  natb lba1 = lba << 8;
  natb lba2 = lba << 16;
  natb lba3 = lba << 24;

```

```
  // il byte pi  significativo deve attivare l'LBA,
  // lba stava comunque su 28 bit
  lba3 = (lba3 & 0x0F) | 0xE0; // 1110-LBA-

  outputb(lba0, disk_lba0);
  outputb(lba1, disk_lba1);
  outputb(lba2, disk_lba2);
  outputb(lba3, disk_lba3);
}

// dai comando al controllore disco
void give_command(natl lba, natb sectors, natb cmd) {
  give_lba(lba);
  outputb(sectors, disk_sectors);
  outputb(cmd, disk_command);
}

// aspetta il disco
void wait_for_disk() {
  natb s;
  do {
    s = inputb(disk_status);
  } while ((s & 0x88) != 0x08);
}

// scrivi un settore sul disco
void write_sector(natb* sector) {
  wait_for_disk();

  // reinterpret_cast per mandare 2 byte per volta (ripetuti a 256 * 2 =
    512)
  outputbw(reinterpret_cast<natw*>(sector), 256, disk_buffer);
}

// leggi un settore dal disco
void read_sector(natb* sector) {
  wait_for_disk();

  // come sopra
  inputbw(disk_buffer, reinterpret_cast<natw*>(sector), 256);
}

// make code salva (1) e carica (2)
const natb save_code = 0x02;
const natb load_code = 0x03;

// indirizzo lba disco
natl lba = 1;

// buffer testo
natb buffer[BUF_SIZE];

// svuota il buffer
void init_buffer() {
  for(int i = 0; i < BUF_SIZE; i++) {
    buffer[i] = 0x00;
  }
}

// cursore buffer testo
```

```
natl cursor = 0;

// sposta il cursore senza uscire dal buffer
inline void mov_cursor(int d) {
  if(cursor == 0 && d < 0) return;

  cursor += d;
  if(cursor >= BUF_SIZE) cursor = BUF_SIZE - 1;
}

// salva buffer testo
void save() {
  give_command(lba, 1, hd::WRITE_SECT);
  write_sector(buffer);
}

// carica buffer testo
void load() {
  give_command(lba, 1, hd::READ_SECT);
  read_sector(buffer);
}

void main() {
  // inizia svuotando il buffer
  init_buffer();

  // vai in un ciclo di lettura
  while(true) {
    natb make_code = get_key();

    if(make_code == esc_code) break;
    if(make_code == back_code) mov_cursor(-1);

    if(make_code == save_code) save();
    if(make_code == load_code) load();

    char c = get_char(make_code);
    if(c != '\0') {
      buffer[cursor] = c;
      mov_cursor(1);
    }

    // aggiorna schermo
    prt_screen(buffer, BUF_SIZE);
    set_cursor(cursor);
  }
}
```

Gli header `keyboard.h` e `video.h` contengono funzioni simili a quelle viste negli esempi precedenti per l'interfacciamento con tastiera e video (ci sono due funzioni video non viste, `prt_screen()` per la scrittura di tutto il buffer video, e `set_cursor()`, che imposta la posizione del cursore hardware agendo su registri specifici).

La scrittura viene effettuata alla pressione del tasto `"1"`, e la lettura alla pressione del tasto `"2"`. Entrambe le operazioni si riassumono fondamentalmente nell'invio di un comando (`give_command()`), che include la scrittura dell'indirizzo LBA (`give_lba()`), e nella successiva scrittura o lettura di un settore (`write_sector()` o `read_sector()`), che comprende di aspettare un certo bit di stato del disco (`wait_for_disk()`). Il bit particolare si può verificare consultando i manuali appositi.

Le funzioni viste finora su periferiche di I/O sono disponibili nella cartella `/code` degli appunti del corso, assieme a vari esperimenti e la definizione completa di tutti gli header usati. Si noti che questi non sempre corrispondono con `libce`, ma spesso riprendono, ridefiniscono o usano (probabilmente in maniera erronea) funzioni e oggetti ivi definiti.

4.2 Caching

Abbiamo detto che la memoria RAM è molto più veloce dei dischi rigidi. Questo è vero, ma non significa che non ci sia comunque un certo dislivello tra la velocità della CPU e la velocità della RAM: un operazione puo' comunque richiedere nel'ordine dei ~ 100 circa cicli di clock.

Per questo motivo si inframezzano fra la CPU e la RAM più memorie, relativamente piccole ma veloci, dette **memorie di cache**.

L'idea è che la RAM in sè è costituita da memoria dinamica (DRAM), quindi a condensatori, relativamente lenta e con tempo di refresh, mentre le memorie di cache vengono implementate con memorie statiche, più veloci ma più costose da realizzare su larga scala (per cui le dimensioni ridotte).

4.2.1 Principi di località

Le piccole dimensioni delle memorie vengono aiutate dalla **località** del codice in memoria: istruzioni che compongono le stesse funzioni avranno istruzioni vicine fra di loro, le strutture definite dal programmatore conterranno dati locali, ecc... In particolare, potremo distinguere fra due **principi di località**:

- **Località temporale:** una volta visto un indirizzo, è probabile che questo o indirizzi ad esso vicini siano visti di nuovo;
- **Località spaziale:** solitamente si accede ad indirizzi vicini fra di loro.

La cache avrà quindi il compito di memoizzare i valori prelevati con frequenza dalla DRAM. Possiamo immaginare che la prima lettura di un dato richiederà il tempo completo di accesso, ma la lettura successiva, ammesso che quel dato sia stato salvato nella cache, richiederà un tempo di accesso significativamente minore.

L'importante è che questo processo sia **trasparente** per la CPU, cioè che questa non si debba preoccupare di quali indirizzi sono stati visti dalla cache e memoizzati e quali no. Il risultato finale è la velocizzazione di un qualsiasi programma senza dover agire in nessun modo sul programma stesso. Di contro, non è detto che il programmatore non possa sfruttare la presenza della memoria cache, cercando di sviluppare algoritmi e strutture dati che rispettano il più possibile i principi di località (tecniche *data driven*).

4.2.2 Cache ad indirizzamento diretto

Vediamo un primo esempio di memoria cache. Abbiamo che lato processore ci arriveranno le linee di byte enable (BE) e le linee di indirizzo (A). Inoltre avremo a disposizione un bus dati (D) di un certo numero di linee.

Vorremo porre fra CPU e DRAM una cache, connessa a quest'ultima dalle linee di indirizzo A. La memoria interna della cache, di dimensione complessiva 64 KB, sarà rappresentata da una serie di blocchi, o **cacheline** da 64 byte.

In fase di lettura, invece di leggere l'unica riga richiesta dal processore, si procederà alla lettura di un certo numero di righe (poniamo 8). Questo significa che per un tempo di lettura di riga di t, ci vorrà un tempo $\sim 8t$ (solitamente meno). La speranza è che queste righe verranno lette successivamente dal processore.

Inoltre, ad ogni blocco di memoria letto dalla cache si dovrà associare dell'informazione riguardo alla posizione in memoria: questa viene contenuta in un altra memoria, dette **memoria delle etichette**. E' quindi più conveniente leggere regioni relativamente più grandi di memoria, in modo da non sprecare *overhead* per piccole quantità di dati.

4.2.3 Principio di funzionamento

La divisione della DRAM sulle cacheline è quindi realizzata giocando sulle scomposizioni degli indirizzi. Si divide ogni indirizzo in tre parti:

- L'**etichetta**, formata dai bit più significativi del bus;
- L'**indice**, formato dai 10 bit centrali (per indirizzare la totalità dei 64 KB di cache);
- L'**offset**, formato dai 3 bit meno significativi di A (per ottenere cache line da 64 byte, cioe $8 = 2^3$ parole quadruple da 8 byte.).

Noto l'offset, l'**indice** verrà calcolato per indirizzare la totalità delle cacheline come stante su un numero di linee tali a:

$$\text{bit}_{\text{indice}} = \frac{\text{dimensione cache}}{\text{dimensione cacheline}}$$

Per ottenere la regione corrispondente ad un indirizzo (il numero di cacheline) si realizza una sorta di *funzione di hash*, prendonendo l'etichetta e usandola come chiave per la regione di dati di indice corrispondente. Inoltre, alla regione selezionata si associa solitamente un singolo bit di validità. Un comparatore fra etichetta e gli n bit piu significativi messo in AND a questo bit di validità ci assicurerà quindi la presenza nella cacheline del dato richiesto, detta **hit/miss**.

4.2.4 Lettura

A questo punto, in fase di lettura, nel caso di hit basterà ricavare una linea di offset dai bit meno significativi di A, e leggere dalla memoria cache a tale offset, all'indice indicato dall'etichetta. Nel caso di miss si dovrà invece svolgere la lettura in memoria RAM, e poi riportare l'informazione nella cacheline di indice giusto della cache aggiornando l'etichetta.

4.2.5 Scrittura

Per quanto riguarda le scritture invece, potremo muoverci in due strade: **write allocate** e **write no allocate**

- **Write allocate:** ci comportiamo in maniera simile alla lettura nel caso di hit. Nel caso di miss, invece, riportiamo il dato in cache.

  A questo punto potremmo pensare di svolgere la scrittura in RAM e in cache contemporaneamente (regola *write-through*), mantenendo entrambe aggiornate.

  Una tecnica più intelligente può invece essere quella di aggiornare il solo dato in cache, e rimandare la scrittura in RAM alla rimozione del dato dalla cache (per

l'introduzione di un nuovo dato allo stesso indice) (regola *write-back*). In questo caso dovremo dotarci di un nuovo bit nella memoria delle etichette, il bit *dirty*, che segnalerà il bisogno di ricopiare il dato in cache nella RAM in occasione del suo deallocamento dalla cache. La difficoltà principale di questo metodo è l'avere un agente che non è la CPU che scrive in RAM, e come vedremo richiede soluzioni tecniche particolari.

- **Write no allocate:** in questo caso ignoriamo le scritture in cache e la sfruttiamo solamente per le letture.

Notiamo che questa cache soffre di problemi di **collisione**: infatti ci sarà un numero di regioni con lo stesso indice ed etichetta diversa, pari alla dimensione della RAM fratto la dimensione della cache.

5 Lezione del 07-03-25

Riprendiamo il discorso della memoria cache. Avevamo che questa è montata fra la CPU e lo spazio di memoria. Più propriamente, questa si trova fra la CPU e il bus.

In questo, può vedere non solo le operazioni sulla memoria, ma anche sullo spazio di I/O. In questo caso, però, dovrà ovviamente comportarsi sempre in maniera *read-through* e *write-through*, quindi effettivamente disattivarsi e lasciare che il processore interagisca direttamente con l'I/O.

Questo è dovuto al fatto che allo spazio di I/O potrebbero accedere e modificare dati dispositivi esterni alla CPU (le interfacce), operazione che invaliderebbe immediatamente qualsiasi cosa venga scritta in memoriaa cache.

Inoltre, ogni operazione di lettura può comportare di per sé un aggiornamento delle interfacce, che comporterà un aggiornamento della memoria, motivo per cui un operazione di caching sarebbe superflua.

Operazione simile varrà effettuata per la memoria video (che non sta nello spazio di I/O). Questa facoltà verra realizzata dalla cache attraverso, probabilmente, *maschere* o *tabelle*.

5.0.1 Cache associative ad insiemi

Avevamo visto come il difetto principale della cache ad indirizzamento diretta è quello delle *collisioni*. Presentiamo un metodo, quello delle **cache associative ad insiemi**, che risolve il problema permettendo di allocare più cacheline allo stesso indirizzo.

Duplichiamo quindi la struttura vista per la cache ad indirizzamento diretto (qui 2 volte), e sruttiamo le uscite hit/miss delle singole memorie delle etichette per pilotare un multiplexer con in ingresso le linee dati delle memorie di cache corrispondenti.

In questo caso a letture allo stesso indice le cache potranno rispondere diversamente (magari la prima in miss e la seconda in hit), e il processore vedrà ritornarsi il dato corretto (in questo caso quello della seconda).

Compito di scegliere quale cache sfruttare nel caso di collisioni è quello del **controllore** di cache (nella cache ad indirizzamento diretto non c'era scelta). La scelta migliore possibile sarebbe quella di scegliere la cacheline al cui i accederà più tardi nel futuro (per mantenere i dati immediatamente utili nella cache).

Chiaramente, visto che non si può prevedere il futuro (o almeno non lo possono fare né la CPU né il controllore di cache), occorre adottare un euristica. Una di queste

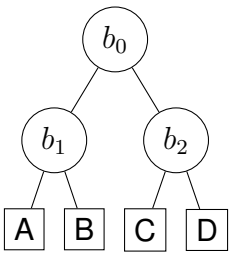
euristiche è la politica **LRU** (*Least Recently Used*), dove si sceglie la cacheline al quale non si accede da più tempo.

Per realizzare tale politica si sfrutta una memoria, che chiamiamo R. Con solo due vie, basterà memorizzare su R l'ultima via usata, e quella su cui scriviere sarà immediatamente l'altra.

Con più di due vie sarebbe necessario mantenere l'ordine degli accessi, cio per n vie ricordare informazione necessaria a controllare $n!$ diverse possibilità. Nella pratica, però, conviene usare politiche approssimate.

5.0.2 Pseudo-LRU dell'80486

Vediamo una di queste politiche approssimate, che gestiva 4 vie attraverso 3 bit $b_0 0$, b_1 e b_2. Si usava un albero binario per la selezione di una delle vie, disposto come:



dove i valori 1 sono i rami a destra, e viceversa i valori 0 sono i rami a sinistra.

In fase di rimpiazzamento, si sceglie la via seguendo l'albero. In fase di accesso, si modificano i b_i in modo da portare la via a cui si è fasso accesso in fondo all'ordinamento che si ottiene visitando l'albero. L'errore può essere dato dal fatto che la via che si trova nello stesso gruppo della via a cui si è fatto accesso potrebbe trovarsi ad un indice più alto del necessario, visto che si abbassa cumulativamente l'intero gruppo aggiornando b_0.

Per cache più grandi si sfruttano sempre algoritmi ad albero di questo tipo, magari tagliando i rami più bassi per lasciare spazio a scelte completamente casuali.

Notiamo poi che le memorie cache di questo tipo incontrano sempre difficoltà quando si fanno accessi ciclici ad indici che si ripetono con un modulo con il numero di vie diverso da zero: ad esempio se si leggono ciclicamente 5 indirizzi che corrispondono allo stesso indice, la cache non riuscirà mai a mantenere tutti e 5 in una delle cacheline delle vie, e quindi ogni accesso comporterà un miss.

5.0.3 Livelli di cache

Nei processori moderni si hanno solitamente più livelli di cache (3 o 4), che crescono in dimensioni e associatività più si vanno a disporre "lontano" dal processore e "vicini" alla RAM. Le cache di livello più basso saranno quindi più veloci ma più piccole, mentre le cache di livelo alto saranno più lente ma più grandi.

Il controllore di cache provvederà a gestire i livelli di cache, effettuando gli accessi controllando a partire dal livello più basso (più veloce) per arrivare al livello più basso, fino alla RAM.

5.1 Interruzioni

La limitazione principale del processore studiato finora è che il flusso di controllo è completamente determinato dal progamma in esecuzione. Attraverso il meccanismo dell'in-

terruzione, il sistema definisce $e_1, ..., e_n$ **eventi**, e il programmatore $r_1, ..., r_n$ **routine** per la gestione di tali eventi. Da qui in poi il processore continua ad eseguire il suo normale flusso di controllo, ma monitorando in qualche modo lo stato di questi eventi. Nel caso uno degli eventi e_i effettivamente si verifichi, la CPU provvederà a sospendere il flusso di controllo attuale e ad eseguire la routine r_i.

Un esempio classico dell'utilità di un meccanismo di questo tipo è dato dalle fasi di stampa che avevamo definito per dispositivi come le stampanti: attraverso l'approcio visto finora dovremmo controllare periodicamente un certo registro di stato per verificare la possibilità di scrivere un nuovo dato in un certo registro di buffer. Questo occupa la CPU con operazioni inutili, che potrebbe saltare se fosse la stampante stessa ad avvertirla di quando è pronta a ricevere un nuovo dato.

L'idea di base è quella di avere una nuova operazione da svolgere in fase di esecuzione di un instruzione da parte della CPU, dopo l'esecuzione dell'istruzione stessa. Ad esempio, potremmo riportarci un bit di valdiità, `READY`, da parte della stampante, e controllarlo ad ogni istruzione per la chiamata di una routine di stampa. La chiamata sarà semplicemente un aggiornamento condizionato a RIP, con scrittura del contenuto attuale di RIP in pila (che è compatibile con le regole di chiamata dei sottoprogrammi a cui siamo abituati).

Un problema di questo approccio potrebbe essere che, se il bit che segnala l'evento non si aggiorna immediatamente, la CPU andrà in un ciclo continuo di arresto dell'esecuzione e inizio di una routine. Una soluzione potrebbe essere dotare della CPU di una *rete di accettazione* della richiesta: il bit di segnalazione dell'evento va in un generatore di impulsi che setta un SR flip-flop. A questo punto la CPU risponde (livello hardware, nella nuova fase di esecuzione appena descritta) con un segnale di reset nel momento in cui riesce a rilevare l'evento e spostarsi nella routine.

In verità la situazione è più complicata: ad esempio potremmo voler ignorare nuovi eventi quando stiamo già cercando di soddisfarne uno. Per questo i processori x86 prevedono un apposito flag, il flag **IF** (*Interruption Flag*), che determina se le nuove interruzioni dovranno essere soddisfatte o meno. Il processore può essere quindi configurato per attivare automaticamente il flag IF in fase di risposta ad una richiesta di interruzione. Per effettuare il corretto ritorno, si usa la funzione `IRETQ`, che ripristina, oltre ad altre cose, lo stato dei flag (che era stato salvato in pila).