1 Lezione del 25-02-25

1.1 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.

1.1.1 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.

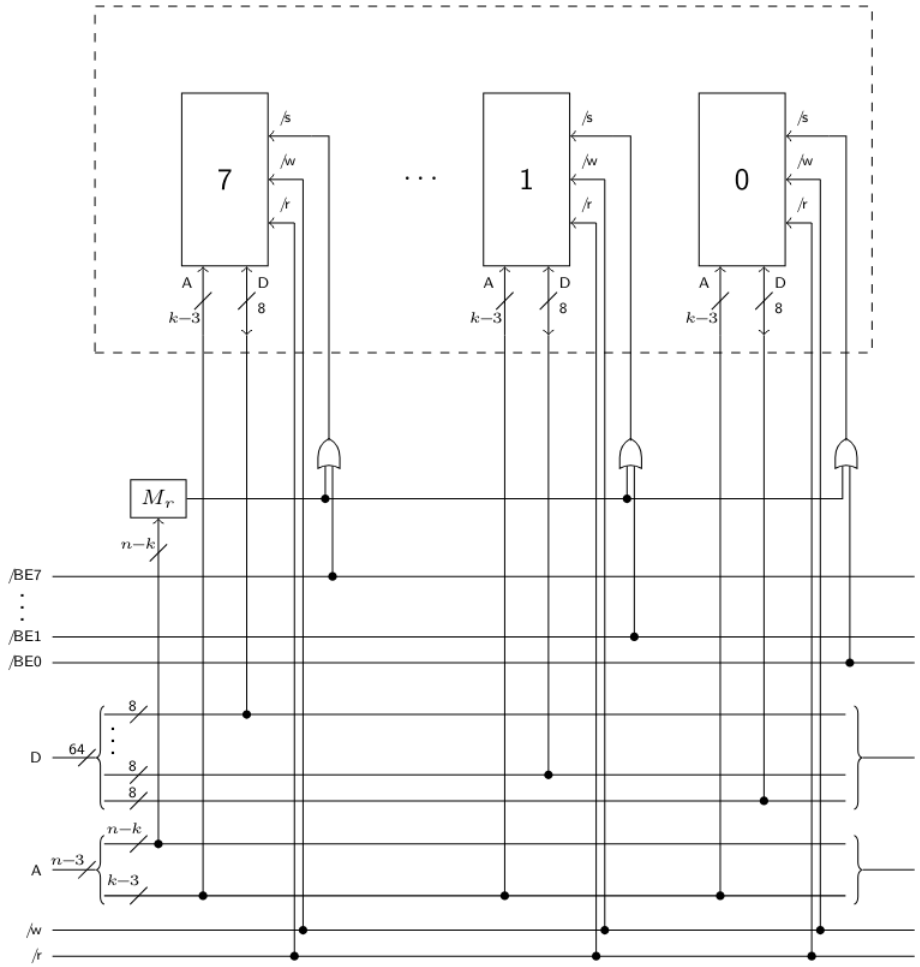
Dalla CPU arriveranno, come abbiamo detto, i **numeri di riga**, i **byte enable**, il **bus dati** e le **linee di controllo**.

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.

La struttura complessiva della RAM sarà quindi la seguente:



dove la rete M_r è quella che si occupa di generare la maschera, prendendo banchi di dimensione 2^k righe.

1.1.2 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.

1.2 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**.

1.3 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:

`0x60`	**RBR**, *Receive Buffer Register* **TBR**, *Transmit Buffer Register*
...	
`0x64`	**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;
    }

    // ottieni carattere e stampa
    char c = get_char(make_code);
    vid::char_write(c);
  }
}
```

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.