# Prova pratica di Calcolatori Elettronici

*C.d.L. in Ingegneria Informatica, Ordinamento DM 270*

10 luglio 2017

1. Siano date le seguenti dichiarazioni, contenute nel file `cc.h`:

```
struct st1 { char vi[4]; }; struct st2 { int vd[4]; };
class cl
{       char v1[4]; char v2[4]; long v3[4];
public:
        cl(st1 ss); cl(st1 s1, long ar2[]);
        cl elab1(char ar1[], st2 s2);
        void stampa()
        {       char i;
                for (i=0;i<4;i++) cout << (int)v1[i] << ' '; cout << endl;
                for (i=0;i<4;i++) cout << (int)v2[i] << ' '; cout << endl;
                for (i=0;i<4;i++) cout << v3[i] << ' '; cout << endl << endl;
        }
};
```

Realizzare in Assembler GCC le funzioni membro seguenti.

```
cl::cl(st1 ss)
{
        for (int i = 0; i < 4; i++) {
                v1[i] = v2[i] = ss.vi[i]; v3[i] = ss.vi[i] + ss.vi[i];
        }
}
cl::cl(st1 s1, long ar2[])
{
        for (int i=0; i<4; i++) {
                v1[i] = v2[i] = s1.vi[i]; v3[i] = ar2[i];
        }
}
cl cl::elab1(char ar1[], st2 s2)
{
        st1 s1;
        for (int i = 0; i < 4; i++)
                s1.vi[i] = ar1[i];
        cl cla(s1);
        for (int i = 0; i < 4; i++)
                cla.v3[i] = s2.vd[i];
        return cla;
}
```

2. Colleghiamo al sistema delle periferiche PCI di tipo `ce`, con vendorID `0xedce` e deviceID `0x1234`. Ogni periferica `ce` usa 32 byte nello spazio di I/O a partire dall'indirizzo base specificato nel registro di configurazione BAR0, sia $b$.

   La periferiche `ce` sono periferiche di ingresso in grado di operare in PCI Bus Mastering. Sono inoltre in grado di eseguire tutte le necessarie trasformazioni da indirizzi virtuali a fisici, utilizzando le stesse strutture dati della MMU del processore, purchè non incontrino bit P a zero durante la traduzione.

   I registri accessibili al programmatore, tutti di 4 byte, sono i seguenti:

   1. **VPTRHI** (indirizzo $b$): parte più significativa dell'indirizzo virtuale di destinazione (sempre 0 nei sistemi a 32bit);
   2. **VPTRLO** (indirizzo $b + 4$): parte meno significativa dell'indirizzo virtuale di destinazine;
   3. **CNT** (indirizzo $b + 8$): numero di byte da trasferire;
   4. **CR3** (indirizzo $b + 12$): indirizzo fisico del direttorio (32bit) o tabella di livello 4 (64bit);
   5. **STS** (indirizzo $b + 16$): registro di stato;
   6. **CMD** (indirizzo $b + 20$): registro di comando.

   Ogni volta che si scrive il valore 1 nel registro CMD, la periferica tenta di scrivere CNT byte in memoria a partire dall'indirizzo virtuale contenuto in VPTRHI, VPTRLO. Gli indirizzi verranno tradotti utilizzando la tabella di corrispondenza puntata dal registro CR3 dell'interfaccia. Se l'interfaccia incontra degli errori durante la traduzione (per es. un bit P a zero), interrompe il traferimento e setta il secondo bit di STS. In ogni caso la periferica invia una richiesta di interruzione al completamento dell'operazione (o perché non ha più byte da trasferire, o perché ha riscontrato un errore).

   Le interruzioni sono sempre abilitate. La lettura del registro di stato funziona da risposta alle richieste di interruzione.

   Modificare i file `io.s` e `io.cpp` in modo da realizzare la primitiva

   ```
   bool cedmaread(natl id, natl quanti, char *buf)
   ```

   che permette di leggere `quanti` byte dalla periferica numero `id` (tra quelle di questo tipo), copiandoli nel buffer `buf`. La primitiva restituisce `false` se il trasferimento è stato interrotto per errori, `true` altrimenti.

   La primitiva deve essere in grado di funzionare sia se il buffer dell'utente è residente, sia se non lo è. Se non lo è, può aggirare il problema eseguendo il trasferimento in un buffer residente intermedio e poi copiando dati da questo al buffer del utente. Il buffer dell'utente potrebbe essere residente solo per alcune parti. La primitiva deve usare buffer intermedi solo per le parti non residenti, eventualmente ordinando più trasferimenti distinti.

   Controllare tutti i problemi di Cavallo di Troia.

   Per descrivere le periferiche `ce` aggiungiamo le seguenti strutture dati al modulo I/O:

   ```
   struct des_ce {
       natw iVPTRHI, iVPTRLO, iCNT, iCR3, iSTS, iCMd;
       natl sync;
       natl mutex;
       bool error;
   };
   des_ce array_ce[MAX_CE];
   natl next_ce;
   ```

La struttura `des_ce` descrive una periferica di tipo `ce` e contiene al suo interno gli indirizzi dei vari registri, l'indice di un semaforo inizializzato a zero (`sync`), l'indice di un semaforo inizializzato a 1 (`mutex`) e un booleano per memorizzare il successo o fallimento dell'ultima operazione.

I primi `next_ce` elementi del vettore `array_ce` contengono i descrittori, opportunamente inizializzati, delle periferiche di tipo `ce` effettivamente rilevate in fase di avvio del sistema. Ogni periferica è identificata dall'indice del suo descrittore.

**Nota**: il modulo sistema mette a disposizione la primitiva

```
natl residentpart(void addr, natl quanti);
```

La primitiva `residentpart()` restituisce la lunghezza in byte del più lungo *prefisso residente* del buffer. Con questa dicitura ci riferiamo al numero di byte consecutivi del buffer che risultano residenti, partendo dal byte di indirizzo `addr`.

Inoltre, nel modulo I/O sono presenti le seguenti funzioni:

```
addr mem_alloc(natl quanti);
void mem_free(addr buf);
```

La funzione `mem_alloc(quanti)` alloca `quanti` byte nello spazio di I/O condiviso e restituisce un puntatore alla zona di memoria allocata (0 se l'allocazione è fallita). La funzione `mem_free(buf)` dealloca una zona di memoria precedentemente allocata con `mem_alloc()`.