I/O Hardware

Per capire come interagiscono i vari componenti dobbiamo usare un modello astratto del controller, così possiamo parlare in generale della comunicazione tra i controller e il processore.

Il problema è che i controller sono molto eterogenei tra loro quindi dobbiamo usare una **astrazione** sia per **l'architettura del sistema** sia per di **dispositivi**.

Il sistema di riferimento:

- **CPU**, **memoria** (RAM) e **dispositivi I/O** sono connessi da un bus che comunicano tra loro.

Dispositivo di riferimento:

- **Interfaccia** (ciò che il processore vede) e una **implementazione interna** (la CPU non la vede). Dentro **l'interfaccia** abbiamo **registri di vario tipo e data buffer**, l'implementazione interna non interessa al processore dato che non riesce ad accedervi.

Interfaccia di un dispositivo

L'interfaccia contiene:

- **Registri dati**: contengono i dati che vogliamo scrivere e leggere, se sono in input li scrive il controller se sono in output la periferica
- **Registri di stato**: vengono usati dal processore per capire cosa sta facendo il dispositivo, sono dei bit che vengono impostati e ognuno di loro ha un significato.
- **Registri di comandi**: sono registri che se vengono attivati (impostato il bit) viene lanciato il comando ad esso collegato.

I dispositivi funzionano in due maniere:

- **Full-duplex**: posso scrivere e leggere in contemporanea (doppio binario)

- **Half-duplex**: possono solo o leggere o scrivere (singolo binario)

In aggiunta a questo abbiamo anche **buffer dati** per immagazzinare momentaneamente dati che **non possono essere** direttamente **passati al SO**, non possono essere attualmente passati al **data register** perchè è **occupato.**

> L'utilizzo dei buffer serve anche per poter sincronizzare i dati in input e in output, ad esempio le cuffie, se la lettura del file audio non fosse controllata e non fossero immagazzinati i dati in un buffer avremmo un audio riprodotto in modo strano.

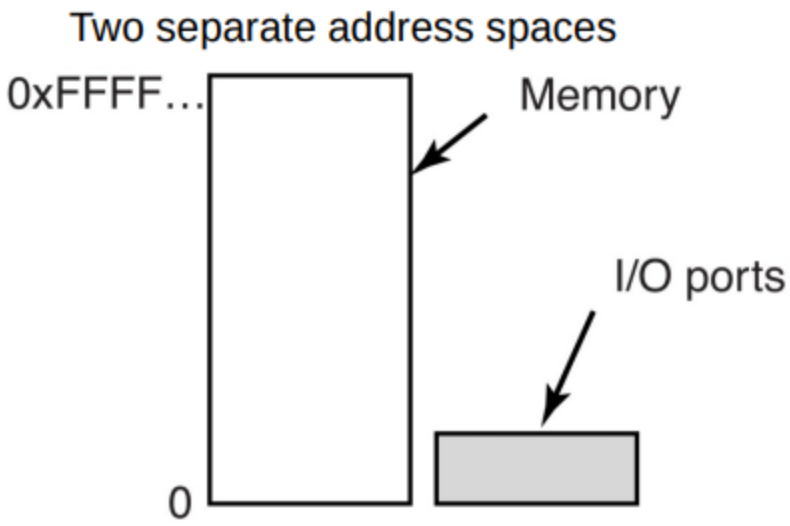
Scrittura e lettura dei registri

Per poter leggere e scrivere dati per una periferica dobbiamo usare abbiamo due approcci:

- **I/O mappato sulle porte →** i **registri del controller** del **dispositivo** hanno una **porta** ad essa assegnata (porta sulla motherboard ovvero pin) e questa mappatura è fatta in una zona riservata della memoria, può essere a 8 o 16 bit, chi assegna questi numeri sono o predefiniti o in fase di boot del sistema. L'insieme di queste porte è chiamato **spazio I/O delle porte, assegnabili** solo **kernel mode** e questo perchè altrimenti un programma di un dispositivo andrebbe a interferire con un altro processo di un altro dispositivo**.**

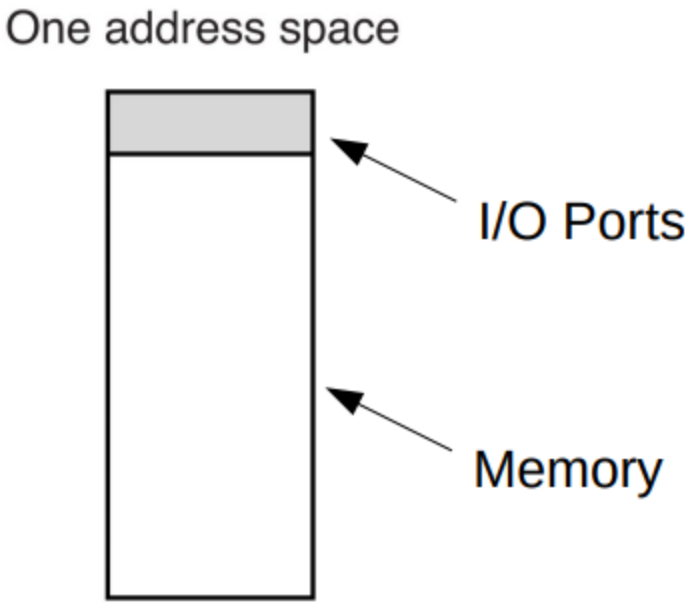
Range di porte	Dispositivo
000-00F	DMA Controller
020-021	Interrupt Controller
040-043	Timer
200-20F	Game Controller
....	**....**

  Per scrivere e leggere dei dati di controller di una periferica dobbiamo usare istruzioni speciali: **IN REG, PORT** (lettura), **OUT PORT, REG** (scrittura). Sono istruzioni privilegiate possibili solo in **kernel** mode.



> 💡 **Mapping su porte**
> A ogni registro di device viene assegnata una porta (pin fisico sulla memoria) e viene modificato il contenuto con istruzioni apposite. viene gestito come uno spazio riservato. Il mapping avviene in questo modo: nella memoria RAM viene assegnato l'indirizzo del registro del dispositivo.

- **I/O mappato in memoria** → i **registri del controller** del **dispositivo** sono mappati in **memoria** ogni controller ha un indirizzo univoco, fatto solitamente al momento del boot. Ora non ci servono istruzioni speciali, essendo dentro la memoria possiamo usare **MOV REG,ADDR** in input e **MOV ADDR, REG** in output.



> 💡 **Mapping su memoria**
> A ogni registro di device viene assegnata un indirizzo e viene modificato il contenuto con istruzioni apposite. Sono all'interno della memoria.

- **I/O ibrido** → possiamo avere un **approccio ibrido,** il quale usa entrambi. Vengono usate perchè a volte bisogna fare così tante operazioni come la scheda grafica che deve dare milioni di pixel, si preferisce usare le istruzioni di memoria molto più veloci.

La forza di usare la mappatura a memoria invece che a porte è che la prima può accedere direttamente ai registri, nel secondo bisogna usare il linguaggio assembler. Un secondo problema è che ci sono diverse versioni di assembler e quindi meno portatile.

> 💡 La mappatura serve per poter **collegare** il **controller** di un dispositivo con la **CPU**, in questo caso abbiamo due mappature, per porte o per indirizzi, nel **primo caso** trattiamo una **porzione di memoria come indipendente** nel **secondo** come se facesse parte della **memoria centrale**. Una volta collegati la **CPU può comunicare con i dispositivi** attraverso istruzioni privilegiate IN e OUT oppure MOV nel secondo caso. La mappatura consiste nel assegnare l'indirizzo in memoria del registro del device e così vengono collegati

La CPU quando esegue una MOV non controlla il PSW, la esegue, se deve controllare la IN e la OUT ci vogliono più cicli macchina e quindi più oneroso e meno efficiente, ma più sicuro.

- Con mappatura di memoria:

```
LOOP: TEST X ; Check if the device is idle by testing if addr. X
 BEQ READY ; If it is 0, go to READY
 BRANCH LOOP ; Otherwise, continue testing
READY:… ; Send to the idle device
```

- Con mappatura di porte:

```
LOOP: IN REG, X ; Read the contents of port X into CPU register
 TEST REG ; Check if the device is idle by testing if REG is 0
 BEQ READY ; If it is 0, go to READY
 BRANCH LOOP ; Otherwise, continue testing
READY:… ; Send to the idle device
```

Come possiamo vedere la mappatura delle porte è molto più pesante il carico di lavoro, controlliamo il contenuto della porta X e lo carichiamo in REG, testiamo REG ed eseguiamo un loop, c'è una istruzione in più che influisce sul carico di lavoro.

Abbiamo parlato molto male di mappatura di porte, ma in realtà ha anche lui i suoi punti di forza: **non ci sono porzioni di memoria I/O sprecati**.

I/O Hardware: come viene svolto

Vengono usati tre approcci distinti:

- I/O **programmato** (aka **polling** ovvero su richiesta)**.**
- I/O guidato dalle **interruzioni.**
- I/O usando il **DMA** (**Direct Memory Access**)**.**

I/O Programmato (PIO o Busy Waiting)

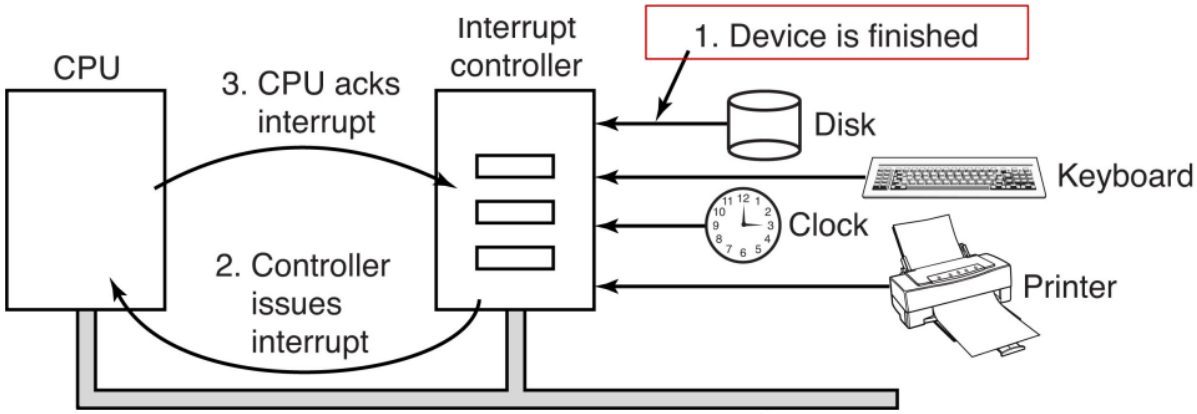
La **CPU ha un ruolo attivo**, chiede al controller di eseguire un I/O, entra in un **loop** nel quale continua a chiedere al **dispositivo** se ha terminato l'I/O, tipicamente ci sono dei **bit di status** nei registri che indicano se il dispositivo è busy o no.

Quando la richiesta di I/O è terminata la CPU può continuare con la prossima istruzione, questo metodo è chiamato anche **busy waiting** (sinonimo di **polling**).

- **Vantaggi**
  - Semplice da **implementare**
  - Perfetto per **piccoli trasferimenti** di data
  - Perfetto se è appropriato che durante l'attesa **non sono possibili altre operazioni** (sicuramente non i SO multi programmati)
- **Svantaggi**
  - La **CPU è impegnata** a controllare lo stato I/O del dispositivo finché non è ready.

I/O Guidato dalle Interruzioni

La **periferica è in grado di mandare un segnale alla CPU** quando ha terminato l'operazione così da **interrompere il processore**, con questa richiesta viene a conoscenza che una periferica ha finito, la CPU nel mentre che non riceve questo segnale **va a fare altro**. Quando il **device controller riceve** il **segnale** dal **dispositivo** può inoltrarlo alla CPU.



1. Il dispositivo **notifica il suo controller** che ha finito
2. L'interrupt controller **se libero gestisce questa notifica** in arrivo
3. Il controller inserisce nel **bus l'id del dispositivo**, invia un segnale elettrico che interrompe il flusso della CPU, a questo punto la CPU termina l'istruzione nell'IS (**Instruction Register**).
4. La **CPU legge** il **numero del dispositivo** che ha generato l'interrupt, e lo **associa all'interrupt vector**, ogni posizione ha l'indirizzo dell'handler (codice di gestione) per il dispositivo.
5. A questo punto la CPU **salva sullo stack tutti i suoi registri** attuali (PC, PSW), passiamo da user mode a kernel mode (**context switch**), la CPU carica il nuovo PC dal interrupt vector.
6. La CPU informa il controller che ha iniziato a gestire questa interruzione.

> 💡 **Interrupt Vector**
>
> [100 | 254 | 333 | ... | ... | ...]
> L'id che arriva del controller è 3, quindi va all'indirizzo 333 dove inizia il programma per la gestione del device con id 3.

In fase di **boot**, vengono **caricati i driver di ogni dispositivo**, così da creare l'interrupt vector, caricando in memoria ogni interrupt handler.

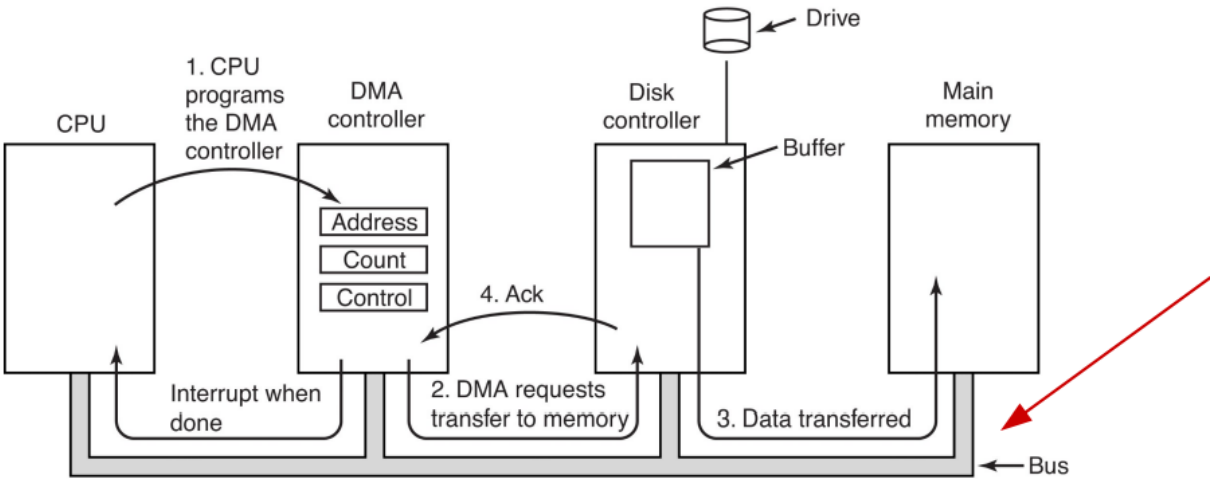
Dopo l'interruzione arriva una istruzione **RETI** (Return From Interrupt) che pulisce il cambiamento dato dell'interrupt caricando in CPU di nuovo i vecchi registri.

I/O usando il DMA

Il **controllore del DMA** (una specie di CPU) serve per gestire lo spostamento dei dati, la CPU delega a lui questi spostamenti.

Il **DMA esegue trasferimenti** dalla **memoria centrale** alla **periferica** senza dover interpellare la CPU ogni volta. **Appena terminato anche l'ultimo blocco** il controller **DMA informa la CPU** del termine del trasferimento.

- **Vantaggi**
  - Non c'è spreco di **tempo della CPU**
  - Ottimo per trasferimenti di **grandi quantità di dati**
- **Svantaggi**
  - Richiede **dispositivi che supportano il DMA**
  - Maggiore **complessità** per controllare il **controller DMA**
  - Alcuni **cicli della CPU** sono usati per fare il **setup del DMA**



Nel DMA abbiamo diversi tipi di registri:

- Registro **dell'indirizzo di memoria**: dove leggere e scrivere dati

- Registro della **conta di byte**: conta quanti byte devono essere letti o scritti in memoria
- Registro di **controllo**: quale registro di I/O usare, qual è la direzione da seguire (se lettura o scrittura), se devo trasferire byte o word, numero di byte da trasferire in un burst.

Come funziona la comunicazione CPU DMA e dispositivo

1. La **CPU programma il DMA** per il trasferimento di dati
    a. La CPU invia anche un comando di “read” al controller del device in questo caso del disco
2. Quando ci sono dati validi nel **buffer del controller** del **disco**, il **DMA inizia il trasferimento**
    a. Il DMA inserisce nel bus l’indirizzo per il controller del disco
    b. Il DMA impartisce l’ordine di lettura al controller del disco
3. Il controller del disco **inizia il trasferimento**, basandosi sulle informazioni passate dal DMA
4. Quando il **trasferimento è terminato** il **controller del disco** manda un **ACK al DMA**
5. Il **DMA decrementa il suo registro** con il **contatore di byte** da trasferire
6. Se il **contatore è maggiore di 0** allora continua dal punto 2 fino al 5, ovvero richiede dati nel buffer e li inserisce
7. Se è uguale a 0 allora manda il segnale alla CPU
8. Quando la CPU si interrompe i dati sono già in memoria centrale

Il metodo visto si chiama **fly-by**: perchè i **trasferimenti non passano dal DMA**, ma il controller del dispositivo scrive direttamente in memoria, ha **l’indirizzo in memoria** e il **modo** (lettura o scrittura). Ci sono delle limitazioni trasferimenti da dispositivo a dispositivo o da memoria a memoria.

Allora hanno creato un’altra modalità ovvero: **fly-through**: i dati che devono essere letti o scritti passano dal DMA, con un buffer interno, i dati poi saranno inseriti o letti dal dispositivo. Questo metodo ha vantaggi ovvero che è molto più

flessibile, ma lo svantaggio è che ci vogliono due cicli di bus, il primo per trasferire dati nel DMA il secondo per trasferire dati in memoria o dispositivo.

Contesa di bus tra DMA e CPU

- **Cycle-stealing**: ovvero il primo che acquisisce il bus può usarlo per il trasferimento, il controller DMA ha sempre una priorità maggiore perchè i dispositivi I/O non possono tollerare ritardi.
- **Modalità Burst**: il controller DMA dice alla periferica ad acquisire il bus, e trasferisce un grande quantitativo di dati tutti assieme, per questo chiamato burst.

Chiaramente i DMA reali sono molto più sofisticati, possono gestire più dati assieme in simultanea, eseguendo un round robin su tutte le periferiche in modo da avere una gestione multipla.

Esempio:

Quanto ci vuole per il DMA inviare 1000 word con t1 per l'acquisizione del bus, t2 per trasferire i dati, t1 è maggiore di t2. Usando i due approcci:

- **Cycle-stealing**: 1000 * [(t1 + t2) + (t1 + t2) + (t1 + t2)] dove il **primo termine** è l'acquisizione del bus e invio del comando del controller, il **secondo termine** è per il trasferimento della parola dal controller alla memoria e il **terzo termine** è per il riscontro dal controller del disco al DMA. Quindi 3000 * (t1 + t2).
- **Modalità burst**: (t1 + t2) + t1 + 1000 * t2 + (t1 + t2) dove il **primo termine** è l'acquisizione del bus per l'invio del comando del controller del disco, il **secondo** è per il controller del disco per acquisire il bus, il **terzo** termine è per il trasferimento di 1000 parole e il **quarto** termine per l'acquisizione del bus per il riscontro.