

Lezione 16

Tecniche di pilotaggio dei dispositivi di I/O

- Le periferiche si differenziano notevolmente fra loro perché c'è
 - Trasferiscono differenti quantità di dati
 - Funzionano a velocità diverse
 - Utilizzano 'formati di dato' differenti

} i dispositivi I/O
- Tipicamente funzionano a velocità più basse rispetto a CPU e Memoria
- Per interfacciarsi con il resto del sistema una periferica ha bisogno di un "controller" o "interfaccia"

i dispositivi I/O hanno bisogno di un controller per comunicare con il sistema

Velocità di trasferimento tipiche per dispositivi di I/O

Dispositivo	Vel. Trasferimento (MB/s)
Tastiera	0.00001
Mouse	0.0001
Modem 56k	0.007
Canale telefonico	0.008
Doppia Linea ISDN	0.016
Stampante Laser	0.1
Scanner	0.4
Ethernet classica	1.25
USB (Universal Serial Bus)(2.5W)	1.5
Cinepresa Digitale	4
Disco IDE	5
CD-ROM 40x	6
Fast Ethernet	12.5
Bus ISA	16.7
Firewire (IEEE 1394) (15W)	50
Monitor XGA	60
Rete SONET OC-12	78
Disco SCSI Ultra2	80
Gigabit Ethernet	125
Conventional PCI 32-bit/33MHz	133
Nastro Ultrium	320
Bus PCI-X (1998)	533
10xGigabit Ethernet (2002)	1250
HyperTransport 1.0 (2001)	6400
100xGigabit Ethernet (2010)	12500
Bus PCI-E (2004)	16000
QuickPath (QPI)	25600
HyperTransport 3.1 (2008)	51200

• USB

- v1 (low speed) 1.5Mb/s=187 KB/s
- v1.1 (full speed) 12Mb/s=1.5MB/s
- v2.0 (hi-speed, 2001) 480Mb/s=60MB/s
- v3.0 (super speed, 2009, 4.5W) 5Gb/s=625MB/s
- v3.2gen1 (superspeed, 2013) 10Gb/s=1.25GB/s
- V3.2gen3 (sperspeed, 2019) 40Gb/s=5GB/s

• SATA

- 1.5Gb/s (2001) → 130MB/s(HDD), 200MB/s(SSD)
- 3.0Gb/s (2001) → 200MB/s (reale)
- 6.0Gb/s (2008) → 400MB/s (reale)
- classica (1980) 10 MB/s = 1.25MB/s

• ETHERNET

- fast (1995) 100MB/s = 12.5MB/s
- Giga (1999) 1Gb/s = 125MB/s
- 10xGiga (2002) 10Gb/s = 1.25GB/s
- 100xGiga (2010) 100Gb/s = 12.5GB/s
- 400xGiga (2017) 400Gb/s = 50GB/s

• PCI-EXPRESS (PCI-E o PCIe)

- v1x (2004) 250MB/s(1lane), 4GB/s(16-lane)
- v2.0 (2007) 500MB/s(1lane), 8GB/s(16-lane)
- v3.0 (2010) 1GB/s(1lane), 16GB/s(16-lane)
- v4.0 (2017) 2GB/s(1lane), 32GB/s(16-lane)
- V5.0 (2019) 4GB/s(1lane), 63GB/s(16-lane)
- V6.0 (2022) 8GB/s(1lane), 121GB/s(16-lane)
- V7.0 (2025 - atteso) 16GB/s(1lane), 242GB/s(16-lane)

$B = \text{bit}$ $B = \text{BYTE}$
 $1 = 10^6$ $1i = 2^{20}$
 $K = 10^3$ $Ki = 2^{10}$
 $G = 10^9$ $Gi = 2^{30}$
 v4 gen 4 (2013) 80Gb/s
 = 10GB/s

velocità

molto diverse tra loro.

Come mi interfaccio con queste velocità?

Architettura del Sistema di I/O

velocità di input/output

- L'architettura del sistema di I/O è...
l'interfaccia fra il dispositivo e il processore

loro, come faccio ad interfacciarmi?

- I dispositivi di I/O sono costituiti da

- Componenti sensori/attuatori (e.g. tastiera, display, disco, ...)
- Componenti elettronici (controller del dispositivo)
- Componenti software (il "device driver")

- Compiti del controller del dispositivo I/O Dispositivo Fisico nel calcolatore

- Eventuale controllo di più dispositivi che appartengono ad una categoria

- Convertire il flusso seriale di bit in BLOCCHI di byte

- Effettuare eventuali correzioni di errore

↳ raggruppati in byte per ottimizz. transf.

- Rendere disponibili i dati alla memoria principale

- o prelevare i dati dalla memoria principale

a seconda se faccio Rx o Tx

- Compiti del device driver del lato SOFTWARE

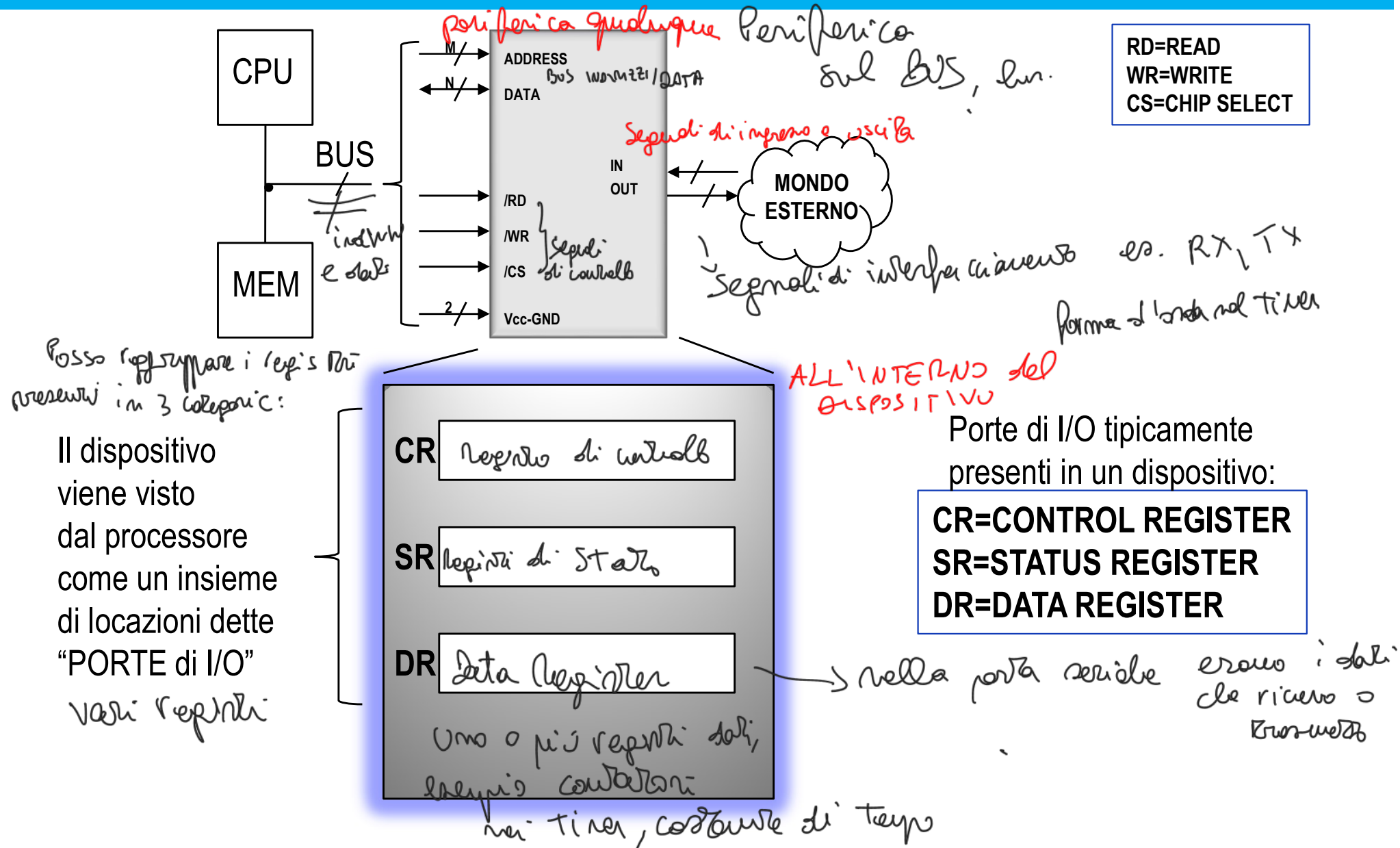
- Inizializzare il controller/dispositivo

- Gestire le operazioni fra controller/dispositivo e processore

via software

delego il trasferimento dei dati ad un altro componente

Visione elettrica e logica del dispositivo



registri venivano nel timer solo da un Risc. di indirizz.

Come parlo con una periferica?

i registri venivano solo tramite un riconosc. di indirizz.

Metodi di comunicazione fra processore e dispositivo

(a) Spazi separati di I/O e memoria

(b) Memory-mapped I/O

(c) Soluzione ibrida

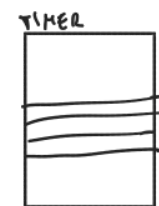
Se non voglio sporcarmi la memoria, posso usare un altro spazio di indirizzamento (metodo a)

① li mappa dirett. in Memoria
② li mappa in uno spazio separato per gli I/O
③ ibrido

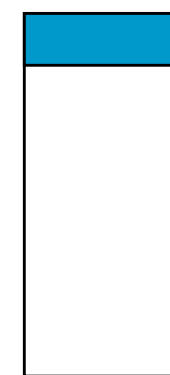


Se servono nuove istruzioni per accedere:
Esempio (architettura x86 antiche):
- `IN AL, [0x60]` → legge un byte all'indirizzo dello spazio di I/O 0x60
ho bisogno di istruzioni particolari per scrivere in I/O

Esempio (architettura RISC-V):
- `LB t0, 0(0x60)` → legge un byte all'indirizzo dello spazio di I/O 0x60
uso `load` e `store` anche mappato in memoria
per comunicare sulle periferiche



(metodo c)



Memoria Porte di I/O

Esempio (architettura x86 moderne):
- `IN AL, 0x60` → legge un byte all'indirizzo dello spazio di I/O 0x60
- `MOV AL, [0x60]` → legge un altro byte all'indirizzo di memoria 0x60 (es. da un altro dispositivo)

Nel caso c, il processore può parlare con i dispositivi usando uno dei due metodi o entrambi

Nei casi a e b, i dispositivi si basano solo sulle porte o sulla memoria rispettivamente

Istruzioni per la comunicazione fra processore e dispositivo

- Se si usano i metodi (a) o (c) per scambiare dati fra processore e memoria si rendono necessarie
 - Istruzioni speciali di I/O (es. Architetture x86)
 - L'istruzione speciale specifica il numero del dispositivo
 - Questo può essere trasmesso "come un indirizzo" sul bus di I/O
 - L'istruzione speciale specifica il comando da dare al dispositivo
 - Questo può essere trasmesso "come un dato" sul bus di I/O
- Se si usa il metodo (b) non è necessario introdurre nuove istruzioni per comunicare con l'I/O *riservo quegli indirizzi*
 - è sufficiente riservare certi indirizzi di memoria, non per mappare RAM ma per mappare indirizzi relativi al dispositivo
 - Letture e scritture a tali indirizzi sono interpretati dalla periferica come comandi
 - Si impedisce all'utente l'uso di questi indirizzi perché risiederanno in una zona riservata allo spazio di indirizzamento kernel
 - Questa tecnica è utilizzabile anche nel caso (c)

Protocolli di comunicazione fra processore e ^{controller} dispositivo

- Il processore ha bisogno di sapere quando
 - Il dispositivo ha completato un'operazione
 - Il dispositivo ha incontrato un errore
- Esistono tre protocolli fondamentali per comunicare con il dispositivo i/o

• **Polling:** chiedere continuamente delego al processore tutti i compiti

- Il dispositivo mette il dato in un "data register" e segnala questo fatto tramite uno "status register"
- Il processore controlla periodicamente lo "status register" controllo sempre lo status register

• **Interrupt:** interrompo processore solo per leggere il dato necessario (es. Tastiera)
efficiente
Ogniqualvolta il dispositivo ha bisogno di attenzione dal processore, interrompe il processore tramite linea dedicata CPU non deve controllare continuamente solo a seguito di un evento

• **DMA/IOP:** processore dedicato

- Il processore inizia l'operazione
- Un processore ausiliario si occupa del trasferimento del dato
- Al processore è notificato che l'operazione è finita con un interrupt dal DMA/IOP verso il processore

flexibile, serve per trasferire dati elevati

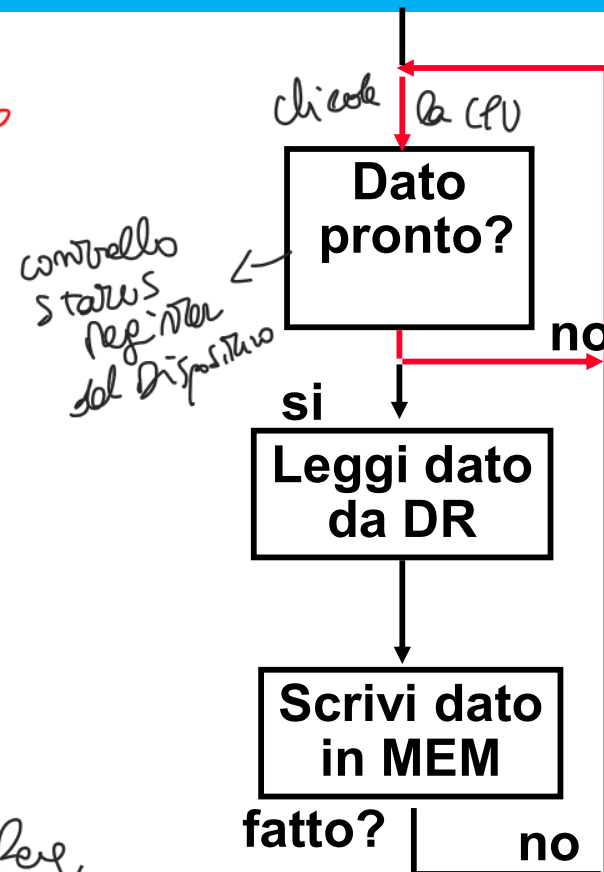
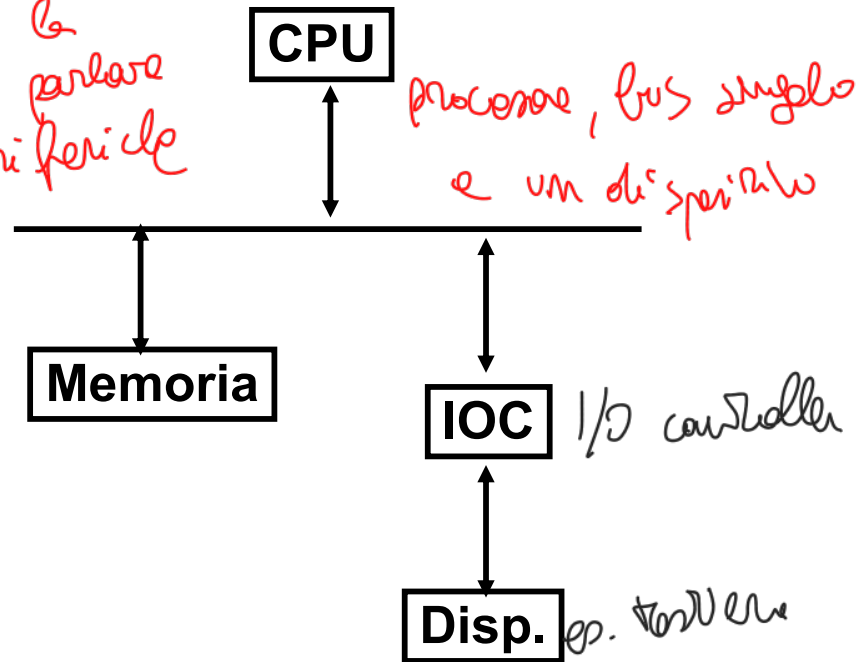
Direct Memory Access

Dma quando ha finito, genera un interrupt

→ Protocollo di POLLING (o di I/O Programmato)

*Scalinato con
c/o tramite polling*

*uso solo la
CPU per parlare
con periferiche*



*Algoritmo del
calcolatore*

**Ciclo di
"attesa attiva"**
*o
"intanto
fai cose"*

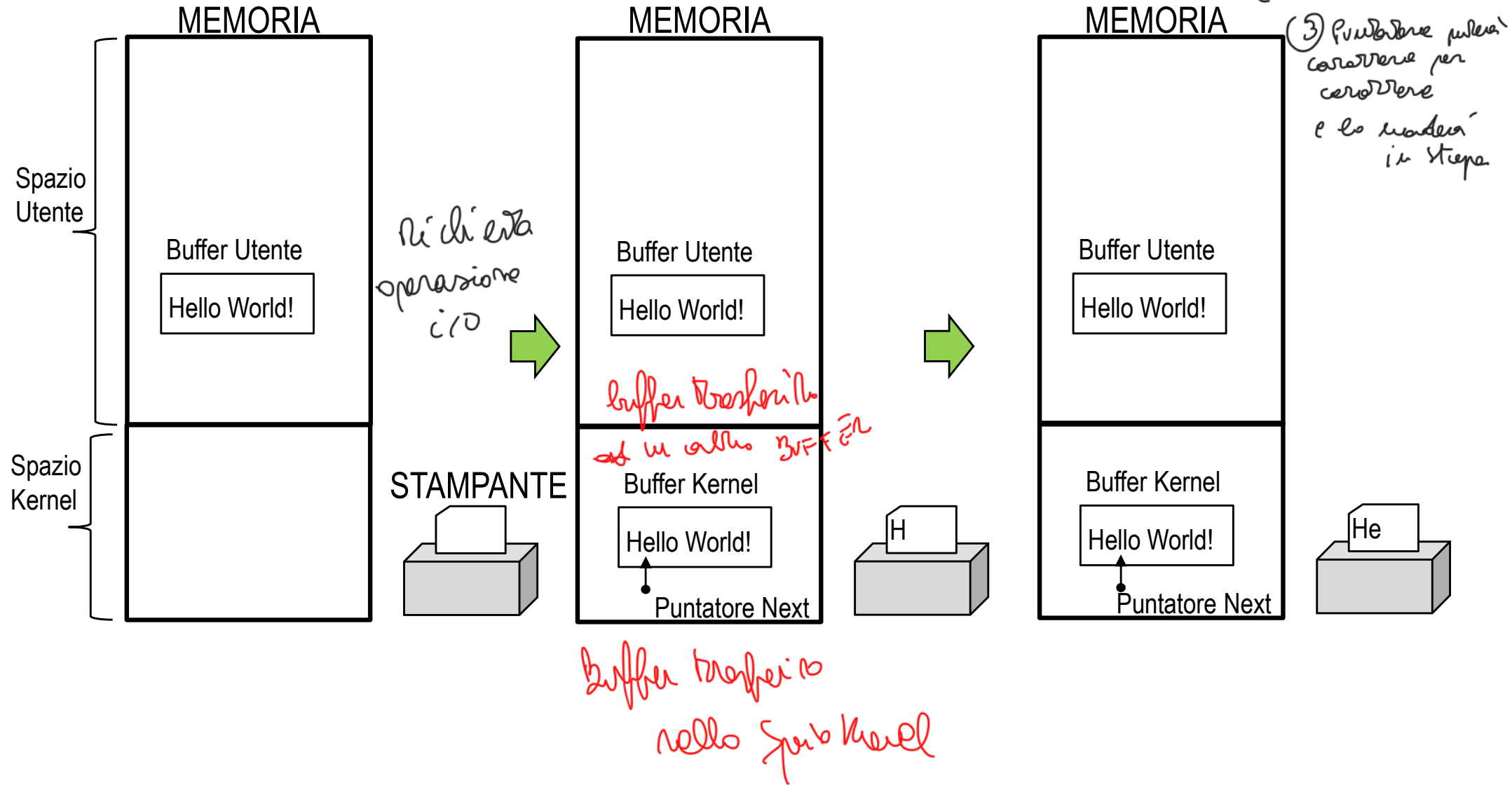
Parlare con periferiche: mette dati nel Data Reg,

- **Vantaggio** *status register e controllo se e' stato ricevuto*
 - Semplice: il processore ha il controllo totale e fa tutto il lavoro
- **Svantaggio**
 - La gestione del polling può consumare molto tempo del processore
 - Trucco: distribuire i controlli di I/O fra codice che effettua altre operazioni

Polling - dettaglio

- La CPU richiede un'operazione di I/O
- Il controller di I/O effettua l'operazione di I/O
- Il controller di I/O attiva un bit nello "status register" -> per dire che ha fatto
 - Il controller di I/O non informa direttamente la CPU ma deve capire la CPU quando è pronto, leggendo nello Status Reg continuamente
 - Il controller di I/O non interrompe la CPU è la CPU che deve capire controllando
- La CPU guarda periodicamente il bit nello "status register"
 - La CPU può attendere oppure ritornare a vedere più tardi / capire che il dato è arrivato e salvarlo

Esempio di Polling: stampa di una stringa



Stampa di una stringa con Polling (Driver)

Faccio ecall
Es. ecall copia stringa
in spazio kernel

```
copy_from_user(buffer, p, count); /* p è un buffer nel kernel */  
for (k=0; k<count; ++k) { /* ripeti per ogni carattere */  
    while (*printer_status_reg != READY); /* attendi il bit di READY */  
    *printer_data_register = p[k]; /* uscita di un carattere */  
}  
return_to_user();
```

Ciclo while
infinito

utente bloccato sulla ecall
finché non si finisce

Notare il punto e virgola !

→ il driver che funziona a polling scorre una carattere per carattere
verifico in STATUS REGISTER se è pronta la periferica
(leppo il bit). Se sì, esco dal ciclo, stampo il carattere
e quando finisco, ritorno il controllo all'utente,
che era stato fermato bloccato sulla ecall

→ Protocollo ad INTERRUPT

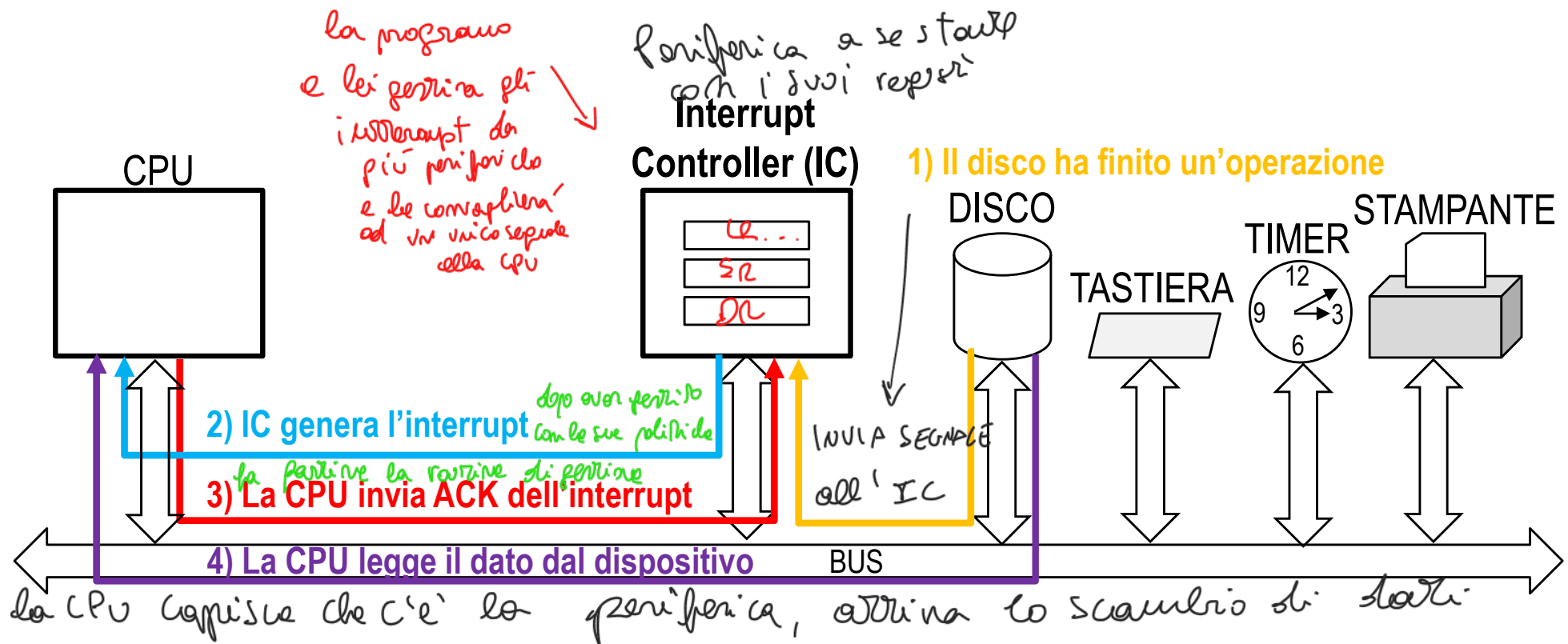
- **Evita l'attesa attiva del processore**

- Evita di fare numerosi controlli sullo stato del dispositivo

- Il controller del dispositivo interrompe una sola volta quando ha bisogno *ba CPU*

Protocollo ad Interrupt: funzionamento con più dispositivi

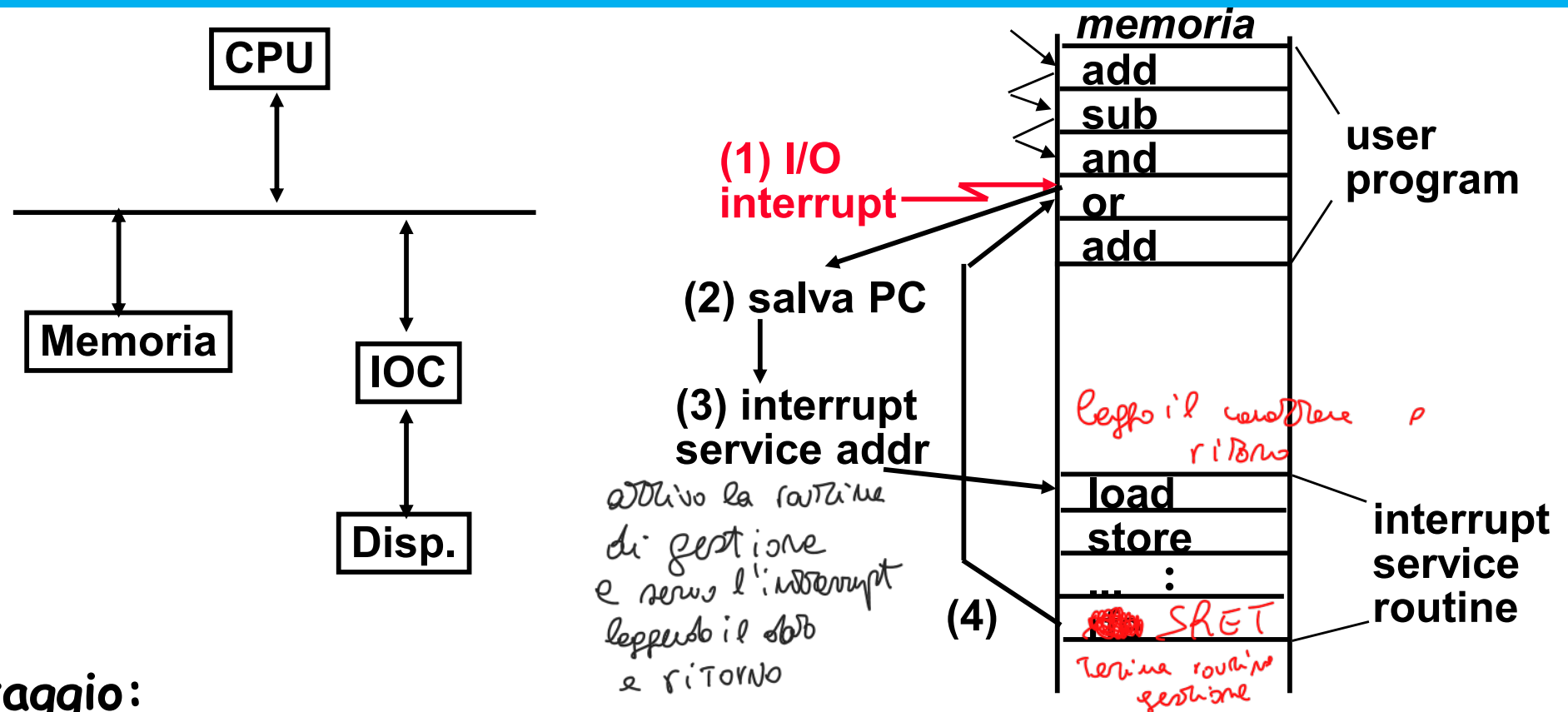
- La CPU dà un comando di lettura (es. dato da disco) e ^{da} questo punto la CPU si occupa di altro... (trascorre tempo)... *non direttamente alla CPU*
- 1) Quando il dato è pronto, il dispositivo invia un interrupt al controller di interrupt (IC), che quindi raccoglie gli eventuali interrupt dai vari dispositivi *è possibile i vari interrupt con la sua politica*
- 2) IC interrompe la CPU *con un solo interrupt*
- 3) La CPU disabilita gli interrupt e notifica IC (ACK) *ho capito che c'è la periferica, attivando lo scambio di dati*
- 4) La CPU legge il dato dal disco e riabilita gli interrupt



Dal punto di vista della CPU...

- La CPU dà un comando di lettura (Vuole leggere sul disco)
- La CPU passa a fare altro lavoro
- Al termine di ogni istruzione guarda se c'è un interrupt pendente
- Se c'è un interrupt pendente $\Rightarrow$ attiva la routine di gestione
 - Salva il contesto
 - Serve l'interrupt: preleva il dato e lo mette in memoria

Trasferimento dati a Interrupt



• Vantaggio:

- Il programma utente è bloccato solo durante il tempo effettivo per trasferire il dato dal dispositivo alla memoria *senza cicli di attesa attivo*

• Svantaggio: è necessario hardware dedicato per *gestire l'interrupt*

- Generare l'interrupt (lato controller di I/O)
- Accorgersi dell'interrupt (lato CPU)
- Salvare lo stato della CPU e ripristinare lo stato dopo l'interrupt

Stampa di una stringa ad Interrupt

Codice di preparazione alla stampa (es. Implementazione di un system call):

1 *copy buffer in user space kernel*
`copy_from_user(buffer, p, count);`
mi fermo per procedere il mio char
`while (*printer_status_reg != READY);`
invo il mio char
`*printer_data_register = p[0];`
il char da processare
`count = count - 1;`
`enable_interrupts();` *abilito gli interrupt*
`scheduler();` *la partine un altro processo, programma*

affido termine
Programma dello scheduler
CPU Fa da
Trasferisco il primo carattere
La chiamata allo scheduler comporta il blocco (cioè la sua deschedulazione dalla CPU) del processo che ha invocato questa system call
Alfido il controllo allo scheduler
aspetto l'interrupt, poi la routine

2 *stampare stampa il mio carattere, arriva l'interrupt*
Routine di servizio dell'interrupt

Se
`if (count == 0) {`
non stampo più i caratteri?
`unlock_user();` *stendo se il la stringa e' stata stampata*
`} else {`
non
`*printer_data_register = p[k];`
combinare a processare caratteri
`count = count - 1;`
`k = k + 1;`
`}`
Routine finisce sempre con, interrupt ricevuto, capito, gestito
`acknowledge_interrupt();`
`return_from_interrupt();` *SR = 7*

Crivono a fine quello che sto facendo
Un interrupt dalla stampante
Inizialmente avrò k=1 quando ha finito

Partine gestione fine
Segue con un altro dell'interrupt, per dire che l'interrupt e' stato capito e gestito

Interrupt

mi sblocca l'utente solo quando
ho stampato tutto

- Per la CPU, l'interrupt è esattamente come un'eccezione salvo che
 - L'interrupt è un evento asincrono, è HW, esterno al codice e utente (!)
 - Non è provocato da istruzioni
 - Non impedisce il completamento di alcuna istruzione
 - Ad esso fa seguito un trasferimento di informazioni
- L'implementazione dell'interrupt è più complicata di quella di un'eccezione, perché
 - Deve consentire di identificare il dispositivo che ha generato l'interrupt l'IC deve identificare il dispositivo che ha generato l'interrupt e comunicarlo alla CPU tramite STATUS
 - Le richieste di interruzione possono avere priorità diversa
 - Le richieste di interruzione provenienti da interrupt diversi possono sovrapporsi (interrupt multipli)

priority e fairness saranno gestite dall'IC

Identificare il dispositivo che ha generato l'interrupt uso Tecnica usata per il Bus

- **Differenti linee per ogni controller** *chi ha generato l'interrupt ?*
 - Usato nei PC
 - Svantaggio: limita il numero dei dispositivi *se ci sono più dispositivi*
- **Software poll**
 - La CPU chiede ad ognuno dei controller se ha generato un interrupt
 - Lento
- **Daisy Chain (Hardware poll)** *← interrompo tutti lo periferico per vedere chi ha generato*
 - Il segnale di Interrupt Acknowledge viene inviato in daisy-chain
 - Il controller è responsabile di mettere sul bus un "vettore"
 - La CPU usa il vettore per identificare la routine di servizio
- **Bus Mastering**
 - Il controller deve richiedere il bus prima di fare interrupt
 - e.g. PCI, SCSI *da priorità ad un dispositivo*

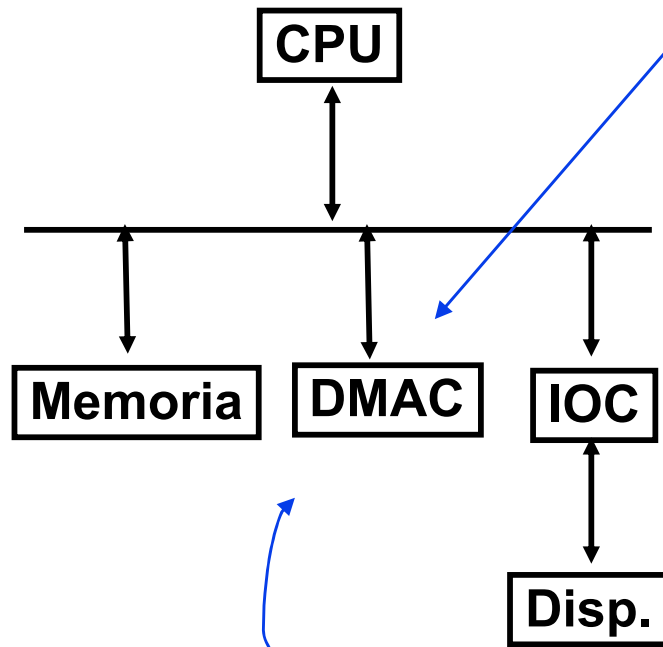
Programma il DMA Controller in modo che
faccia trasferire dati memoria e DISK in maniera
diretta

→ **Direct Memory Access (DMA)**

DMA programma il Controller ecc.

Sul bus ha movimento periferico

(1) Il DMAC si comporta da TARGET



Si occupa del
trasferimento
tra memoria
e controller I/O

1) La CPU dice al DMA Controller: CPU programma il DMAC

- Indirizzo iniziale della zona di memoria coinvolta da trasferire
- Indirizzo del controller del dispositivo indirizzo porta et al ecc
- Direzione (Read/Write) del trasferimento
- Quantità di dati da trasferire 1GB, 1kB ecc
- Infine la CPU dà lo "start"

• La CPU può svolgere altro lavoro

2) Il DMA Controller si occupa del trasferimento cioè genera tutti gli indirizzi coinvolti es. due trasferimenti da 1000 a 2000, due generali da 1004, 1008...

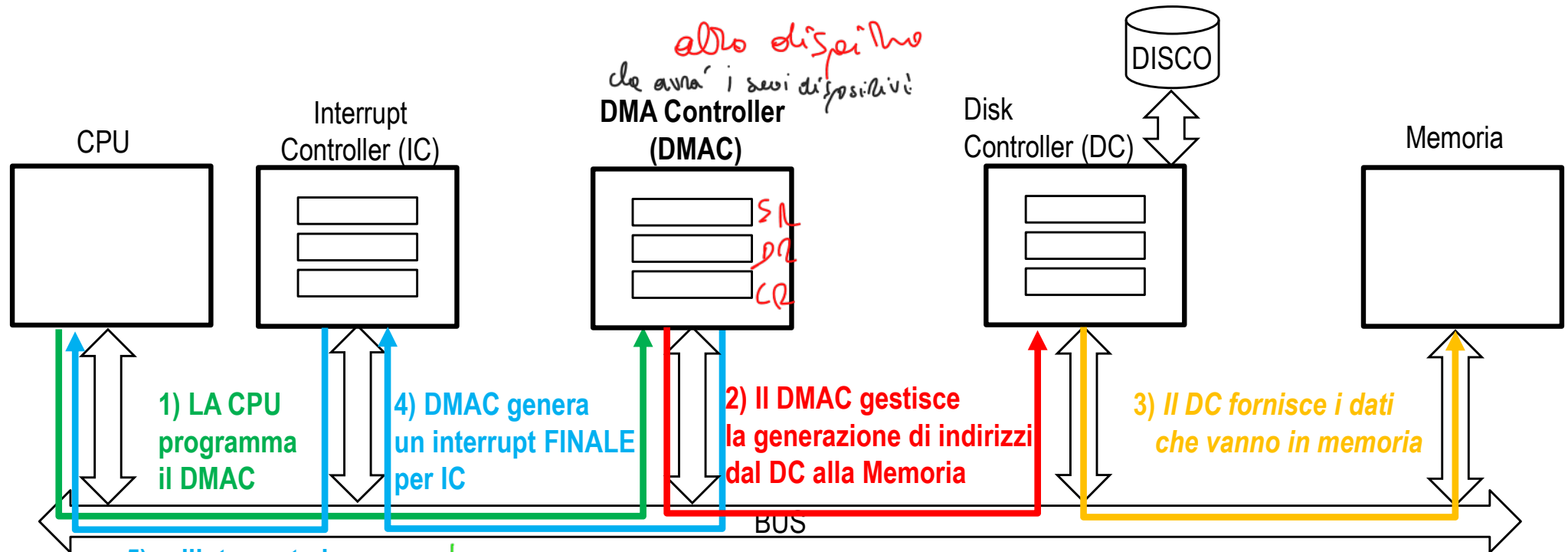
- genera tutti i necessari segnali per indirizzare la periferica e la memoria e per gestire l'handshake per i trasferimenti

• Il DMA Controller manda un interrupt quando ha finito alla CPU

(2) Il DMAC si comporta da INITIATOR

il DMA è un processore più semplice il cui lavoro è quello di spassare gli indirizzi e trasferire dati nella direzione specificata

DMA - dettaglio



5) ...l'interrupt viene inoltrato alla CPU

de lo
prozi'sce
e l'utente
prosegue con
il suo
programma

ok, ho fatto
il trasferimento
da disco a memoria

Lepp' da 1 e trasferisci a disco
Lepp' da 2 e trasferisci a memoria
come CPU fa load e store continuamente
mentre la CPU fa quello che vuole

DMA viene programmato inizialmente dalla CPU
e spedisce i comandi al disk controller.

e fa le load/store tra memoria e disco

Stampa di una stringa con DMA

quando ha finito, il DMA genera un interrupt e IC ferma il processore

Codice di preparazione alla stampa (es. Implementazione di un system call):

(1)

segue
`copy_from_user(buffer, p, count);`
`setup_DMA_controller();` *passa controllo*
`scheduler();` *la CPU fa altro*

La chiamata allo scheduler comporta il blocco (cioè la sua deschedulazione dalla CPU) del processo che ha invocato questa system call → il controllo viene quindi passato al sistema operativo che schedula un altro processo

(2)

Routine di servizio dell'interrupt

ricevo solo in fondo
ok ho ricevuto l'interrupt
`acknowledge_interrupt();`
`unblock_user();` *> sblocco utente, ho ricevuto solo in fondo*
`return_from_interrupt();`
SET

ok ho ricevuto l'interrupt, sblocco

DMA Controller (DMAC) *Spostamento 1, 2, 3D degli indirizzi*

- Il DMA può supportare diverse modalità di indirizzamento
 - 1D, ovvero un singolo registro per gli indirizzi *Spostamento lineare degli indirizzi*
 - 2D, ovvero due registri per l'indirizzamento *es. dove andare degli indirizzi*
 - 3D, ovvero tre o più registri per l'indirizzamento

• Esempio di indirizzamento 2D

- Utile ad es. per prelevare i payload dai pacchetti Ethernet

*il DMA deve fare spostamento indirizzi solo
quelli, voglio solo
estrarre
payload*

