05 - INPUT E OUTPUT

INTRODUZIONE

La **gestione** dell'**IO** comprende la sequenza di operazioni che il processore deve eseguire per

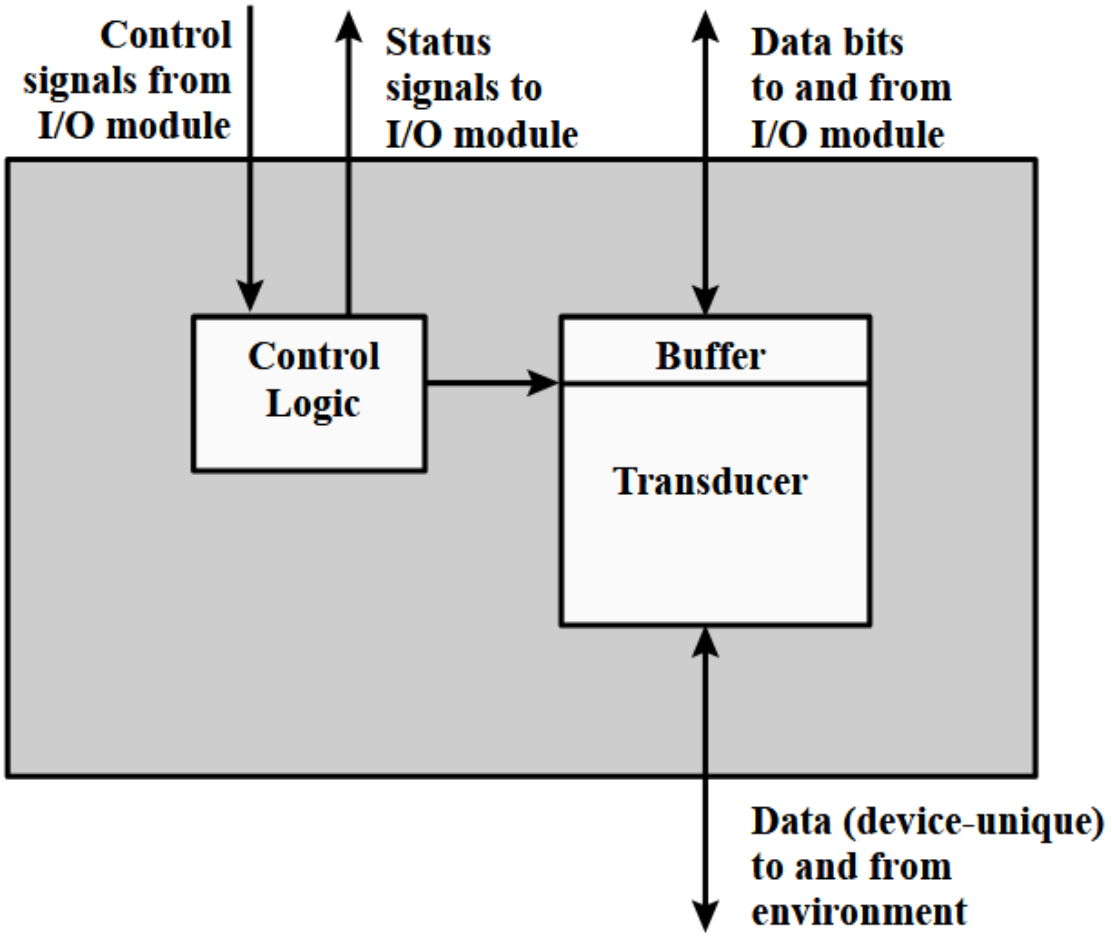
- *Leggere* dati da un dispositivo esterno.
- *Scrivere* dati su un dispositivo esterno.

Esistono *tipi diversi* di IO:

- **Human readable**
- **Machine readable**
- **Communication**

LOGICA DI CONTROLLO

Le operazioni da compiere possono essere molto *diverse* a seconda del *dispositivo connesso*. Per garantire che i dispositivi riescano a comunicare con l'elaboratore, la loro comunicazione deve avvenire attraverso un **protocollo di I/O** *predefinito*.



MODULO I/O

Il modulo I/O costituisce l'**interfaccia standard** con cui il dispositivo I/O comunica con l'elaboratore.
E collegato al dispositivo da un lato, e all'elaboratore attraverso il *system bus* dall'altro.

Il modulo svolge le seguenti funzioni:

- **Controllo** e **temporizzazione**.
- **Comunicazione** con il *processore*.
- **Comunicazione** con il *dispositivo*.
- **Data buffer**.
- **Controllo** degli **errori**.

I/O PROGRAMMATO

Nel **programmed I/O** la **CPU** si fa carico diretto dell'*esecuzione delle istruzioni* per la lettura e scrittura da e per un dispositivo di output.
Ciò avviene attraverso il modulo I/O tramite 4 tipi di comandi:

- **Controllo**
- **Test**
- **Lettura**
- **Scrittura**

ACCESSO AL DISPOSITIVO

L'accesso ai dispositivi può essere implementato in due modi.

MEMORY MAPPED

- I dispositivi I/O sono collegati con le *stesse linee bus* usate anche per la *memoria*.
- I *data buffer* dei moduli I/O sono raggiungibili con dei *normali indirizzi* di memoria.
- Il processore vi può accedere (in lettura e scrittura) con le *stesse istruzioni* con cui normalmente accede in *memoria*.

ISOLATED

- I dispositivi I/O sono collegati con *linee di controllo* e *linee dati* **dedicate**.
- I dispositivi hanno uno *spazio di indirizzi proprio*.
- Il processore deve usare *istruzioni dedicate* per l'accesso ai dispositivi.

LETTURA E SCRITTURA

Quando la CPU deve *accedere* in lettura o scrittura a un dispositivo I/O, ne **controlla lo stato**:

- Se il dispositivo è *pronto*, vi può accedere,
- altrimenti deve *attendere*.

Nel caso della **LETTURA**, i dati vengono *prelevati dal data buffer* e caricati *in memoria*.
In caso di **SCRITTURA**, vengono *prelevati dalla memoria* e *scritti nel data buffer*.

BUSY WAITING

Sia in lettura che in scrittura, la CPU deve **attendere** che il dispositivo sia *pronto*, ma i dispositivi possono essere *molto più lenti* del processore che, mentre attende, **non può** fare *altro* : è **BUSY WAITING**.
Questo ovviamente è un grande spreco di cicli di clock.

PRO E CONTRO DELL' I/O PROGRAMMATO

PRO:

- Molto *facile da realizzare*.

CONTRO:

- Problema *busy waiting*.
- *Inefficienza* crescente con la disparità fra la velocità della CPU e quella del dispositivo.

NOTA: il tempo di risposta può essere veloce, ma solo se il dispositivo è pronto.

I/O A INTERRUPT

Una possibile soluzione alternativa all'I/O programmato è data dalla *gestione a interrupt*:
il processore *non* sta in *attesa* che il dispositivo sia *pronto*, ma è il dispositivo ad "*avvertire*" la CPU quando è *pronto*.

INTERRUZIONI

Logica della soluzione:

- Il processore avverte il dispositivo I/O che deve effettuare un'*operazione I/O*.
- Il *processore* torna a *svolgere le sue operazioni*.
- Quando il *dispositivo è pronto*, **avverte** la CPU.
- A questo punto il processore **interrompe** l'operazione *attuale* ed esegue quella di I/O.
- Terminata questa, la CPU *torna al processo interrotto*.

Chiaramente, il modulo I/O deve essere in grado di generare il *segnale* che *avverte il processore* che il *dato è pronto* (**IRQ**).
Il processore deve:

1. *Interrompere* quello che sta facendo (**Salvataggio Contesto**).
2. Passare a *leggere il dato* pronto (**RSI**).
3. *Ritornare* al processo che stava eseguendo prima (**Ripristino Contesto**).

INTERRUPT REQUEST

Per gestire l'**IRQ** (*Interrupt Request*) dobbiamo aggiungere una nuova linea all'elaboratore che *dal modulo I/O* arrivi *ad un pin* della *CPU*. In caso di *più dispositivi*, possiamo collegare le linee con una *porta OR*.

SALVATAGGIO CONTESTO

Quando il processore riceve il segnale *IRQ* deve **interrompere** l'esecuzione del processo corrente e *gestire l'I/O*: è quindi necessario **salvare** i *dati in uso* e il *PC* per poter tornare al processo una volta terminato l'I/O (procedura simile alla chiamata a subroutine).

In particolare, il processore salva (via *hardware*):

- Il registro *program counter*.
- Il registro con i dati della *ALU*.
- Il registro di stato *CPSR*.

I dati del *programma sospeso* possono essere salvati

- Via **software**, copiandoli nello stack.
- Via **hardware**, utilizzando più *set di registri*. Vedi > SOLUZIONI HARDWARE EFFICIENTI

RSI: ROUTINE DI SERVIZIO

Il suo scopo è quello di effettuare il prima possibile l'*operazione di I/O* sul dispositivo che si è dichiarato ready.

L'**RSI** è un programma *software specifico* scritto per la gestione di un particolare dispositivo I/O. In caso siano presenti dispositivi diversi, avremo più RSI.

Per cui è necessario capire anche *quale dispositivo* ha generato l'IRQ per poter *eseguire l'RSI corretta*, ed eventualmente scegliere quale servire in caso più dispositivi siano pronti.

A seconda dell'*architettura*, queste operazioni possono essere svolte via *software* o via *hardware*.
(Farlo via hardware è più veloce, ma più costoso/complicato da realizzare)

- Via **software** andiamo a leggere il registro di *status* di tutti i dispositivi per capire chi ha generato il segnale di *IRQ* (**software polling**).

- Via **hardware** possiamo avere una *linea IRQ* per *ciascun dispositivo* oppure implementare soluzioni più efficienti (vedi sotto).

GERARCHIA INTERRUZIONI

Il processore riceve costantemente *IRQ*, per cui è possibile che:

- Quando andiamo a controllare chi ha chiamato l'IRQ scopriamo che ci sono *più dispositivi pronti* da servire.
- Mentre è in esecuzione una *RSI*, arriva un'*altra richiesta IRQ*.

Per cui è necessario scegliere *chi servire prima*, gestendo in qualche modo le **priorità**:

- Via **software** la gerarchia è imposta dall'*ordine* con cui viene effettuato il *software polling*: il *primo* dispositivo trovato *ready* viene *servito*.
- Via **hardware** viene realizzato un *circuito apposito* che gestisce le priorità e *blocca* le richieste *IRQ* da parte di dispositivi *meno importanti* di quello attualmente servito.

SOLUZIONI HARDWARE EFFICIENTI

BANKED REGISTERS (SALVATAGGIO CONTESTO)

Potremmo avere *più* **set di registri**: quando dobbiamo eseguire una RSI, usiamo un nuovo set.
La RSI scrive pertanto in questi nuovi registri, mentre i *dati del programma* restano *non modificati* nel *set precedente*.
Terminata la RSI, si ritorna al set precedente.

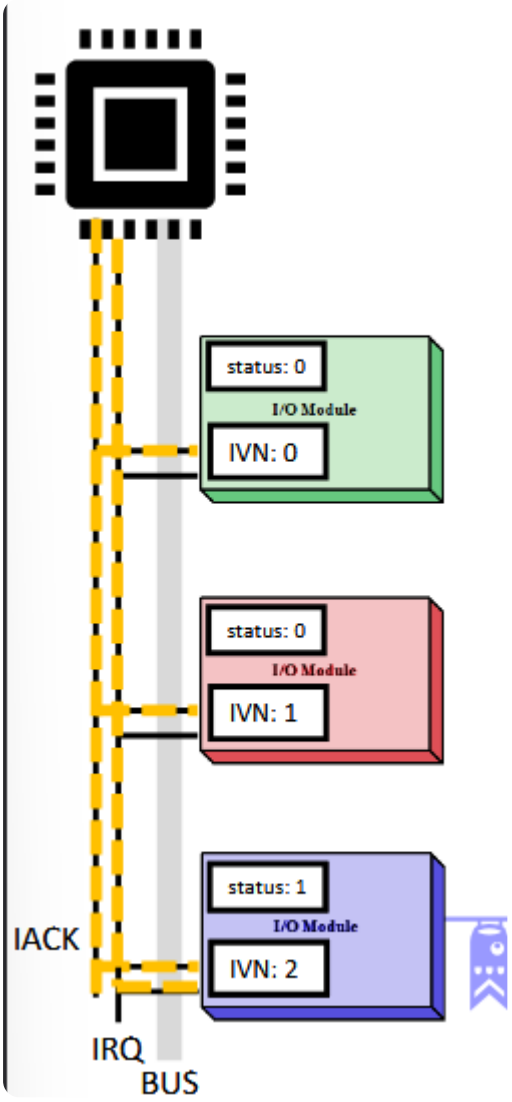
Dato che una RSI può essere *interrotta* solo da IRQ *più prioritari*, ci basta avere tanti *set* quanti il *numero di priorità*.

INTERRUPT VECTOR NUMBER (IDENTIFICAZIONE)

Il processore, in risposta al segnale IRQ, risponde con un *segnale* **IACK** (IRQ Acknowledge).
Il dispositivo invia sulla linea dati il suo *codice identificativo*, che indica la *riga* da guardare in una **tabella** che contiene l'indirizzo di *tutte le RSI*.

- Un dispositivo pronto genera un segnale *IRQ*.

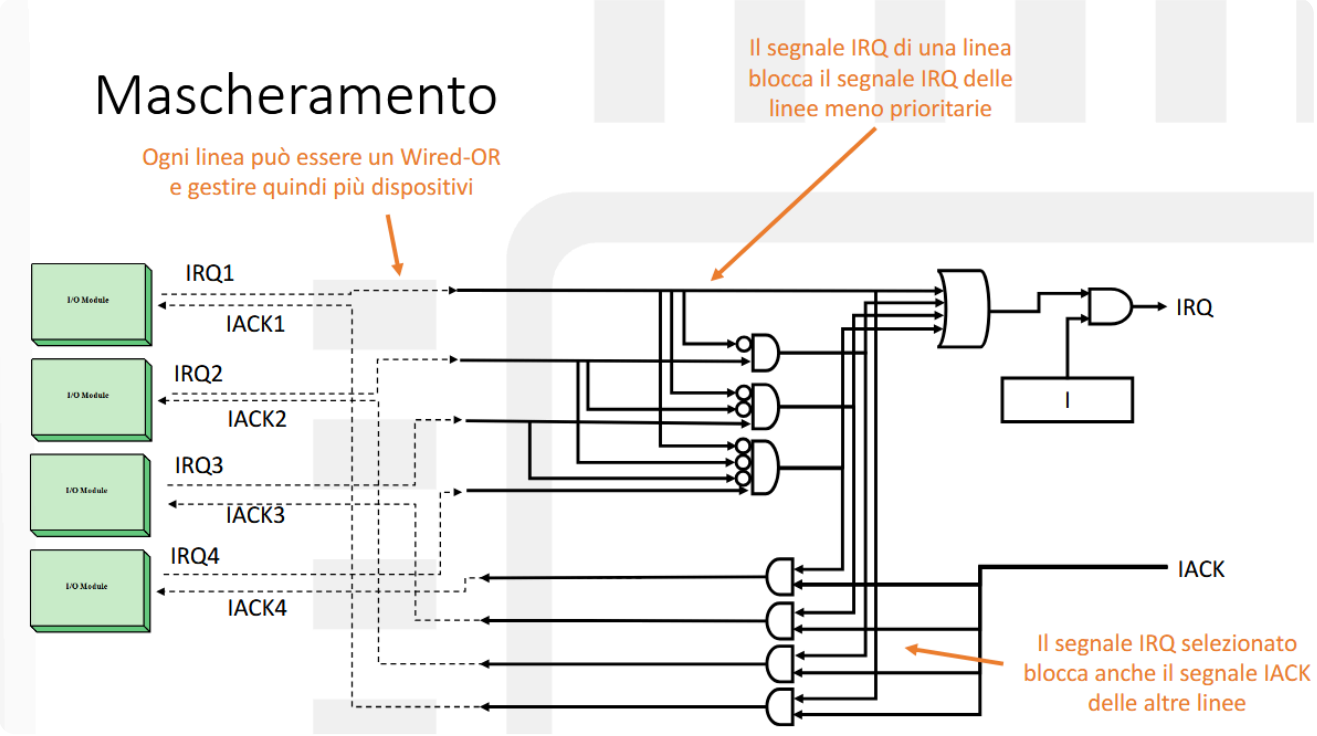
- Quando il processore è pronto a gestire l'interruzione, genera il segnale *IACK* e lo trasmette a tutti i dispositivi.
- Solo quello *ready* risponde, inviando il suo **IVN** (*Interrupt Vector Number*).
- L'*IVN* viene usato come *offset* per *leggere* un valore della **IVT** (*Interrupt Vector table*).
- Il valore letto, cioè **IVT[IVN × 4]**, è l'*indirizzo in memoria* della *RSI da eseguire*.



MASKING E DAISY CHAIN (GESTIONE PRIORITA')

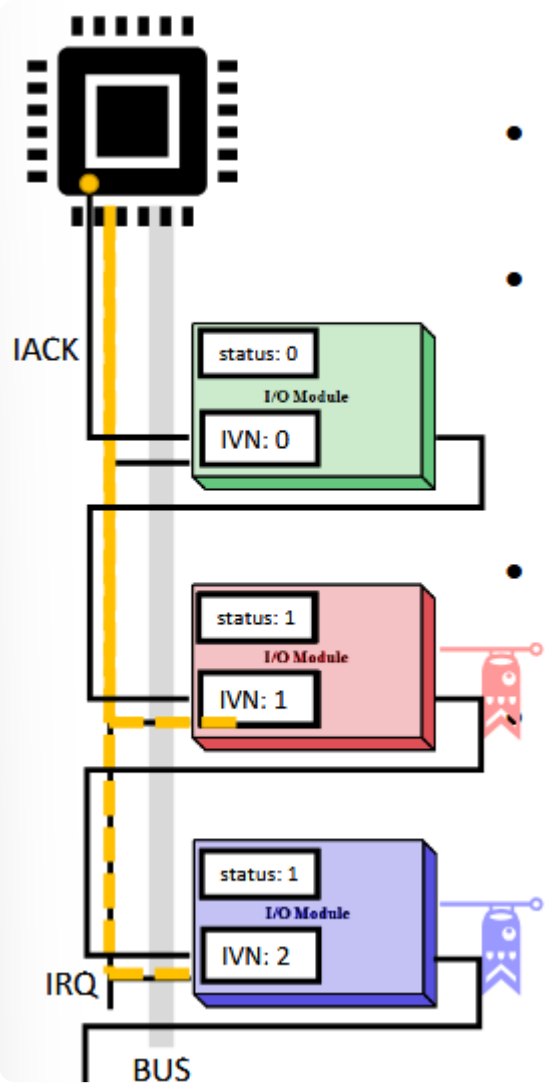
MASCHERAMENTO

Se abbiamo più linee di IRQ che entrano nel processore, possiamo fare in modo che un IRQ a *priorità maggiore* **mascheri** i segnali di IRQ a *priorità minore* con delle semplici porte logiche.
Con un meccanismo analogo possiamo anche *bloccare* l'invio del segnale *IACK* della CPU sulle linee a priorità minore di quella attualmente servita.



DAISY CHAIN

Possiamo collegare i dispositivi in **Daisy Chain**: la linea del segnale *IACK* entra in ogni dispositivo e da questo prosegue verso il successivo.
In questo modo, se il dispositivo ha *generato l'IRQ* blocca la propagazione del segnale e invia al processore il suo *IVN*, altrimenti il segnale passa al dispositivo successivo.
La **priorità** è quindi data dall'*ordine* con cui sono collegati i dispositivi.



INTERRUZIONI IN ARM

In ARM le interruzioni sono chiamate **eccezioni** e organizzate in *6 livelli di priorità*.
(Di questi, *2* dedicati a eventi *I/O*: *IRQ* e *FIRQ Fast IRQ*).

La loro gestione è resa efficace dall'uso di *interrupt vettorizzati* e *banked registers*.
Inoltre, sono implementati anche altri accorgimenti:

- Istruzioni che richiedono *più di un ciclo* possono essere *spezzate*.
- La IVT non contiene indirizzi, ma un'*istruzione* (tipicamente branch) in modo da risparmiare tempo.
- Se al termine di una RSI c'è già un'altra *IRQ pending* il *contesto non viene ripristinato*, ma viene lasciato nello stack e si passa *direttamente alla seconda RSI*.

I/O CON DMA

DATA RATE

Un altro parametro molto importante nella gestione dell'I/O è il **DATA RATE**, ovvero quanti dati al secondo possiamo trasferire.

In un sistema a *interrupt*, ogni esecuzione della *RSI* gestisce una quantità limitata di dati. Consideriamo per esempio una RSI che *impiega* $10\mu s$ per essere *eseguita*: potrà essere ripetuta 100 mila volta in un secondo.
Poniamo che gestisca 16 bit ad esecuzione, il **data rate** sarà di $200\ \mathrm{KB/s}$.

NOTA: A questa velocità, il processore sarà *costantemente impegnato* a gestire gli IRQ senza poter fare altro. Inoltre, se il dispositivo dovesse produrre più dati di quelli gestibili, una parte di essi andrebbe persa.

Il problema è quindi che il processore viene *interrotto* ogni volta che *una piccola parte dei dati è pronta*: per il trasferimento ad alta velocità di dati da/verso la memoria, il sistema a interrupt non va bene.

DMA CONTROLLER

Aggiungiamo all'elaboratore un **DMA Controller**.
E' una componente che ha **accesso** ai *dispositivi I/O* e alla *memoria* tramite il bus: il processore comunica *quanti* dati deve leggere, da *quale dispositivo* e la *posizione iniziale* dove scriverli in memoria e il **DMA** si occupa del loro *trasferimento*, avvertendo la CPU con un IRQ una volta terminato.

NOTA: Un DMA controller è di fatto un *piccolo processore*:

- Contiene un **registro** per il *numero di dati* da trasferire e uno per l'*indirizzo di memoria*.
- Ha un modulo di *Control Logic*.
- Può *controllare* diversi dispositivi di *I/O*.

Durante lettura/scrittura in memoria il DMA controller:

- Riceve una *richiesta* dalla CPU e si pone in *busy waiting* sul dispositivo.
- Appena pronto, *trasferisce il dato*, *decrementa* il *contatore* dei *dati* rimanenti e aggiorna il puntatore in memoria.
- Quando il contatore è zero, *avverte* il processore con un *IRQ*.

ACCESSO AL BUS

Per trasferire i dati, il DMA deve *accedere* al **BUS**, che però è *condiviso* con il processore. Quando il DMA deve utilizzare il bus, avverte il processore con un segnale **HOLD request** e il processore *consente* l'accesso con un segnale **HOLD ACK**.

Può capitare che il bus serva contemporaneamente sia alla CPU che al DMA: in tal caso, tipicamente, si da *priorità* al *DMA* perchè la perdita di dati è considerata più grave.

Quando il *bus* è *occupato* dal DMA, il processore *perde un ciclo*: si parla di **cycle stealing**. Per evitare che la CPU resti bloccata troppo a lungo, il trasferimento viene fatto *un dato alla volta* e l'*arbitraggio del BUS* è ripetuto a *ogni ciclo* di memoria.

Con questa tecnica possiamo pensare di **condividere** il *bus* e la *memoria* anche con processori supplementari:

- Processori *ausiliari* (es. specializzati per la grafica).
- Processore *paritari* multipli (sistemi **multiprocessore**) che permettono il calcolo parallelo e la ridondanza.

PRO E CONTRO

PRO

- Permette il *trasferimento ad alta velocità* di dati dai dispositivi alla memoria e viceversa.
- Il processore è *impegnato* **solo** quando il *trasferimento è completato*.

CONTRO

- Sistema *complesso*.
- Richiede *hardware dedicato*.
- Richiede *software dedicato*.
- Impegna il *bus* di sistema.