Appunti IoT Systems

Capitolo 9

Cyber-Physical Interface

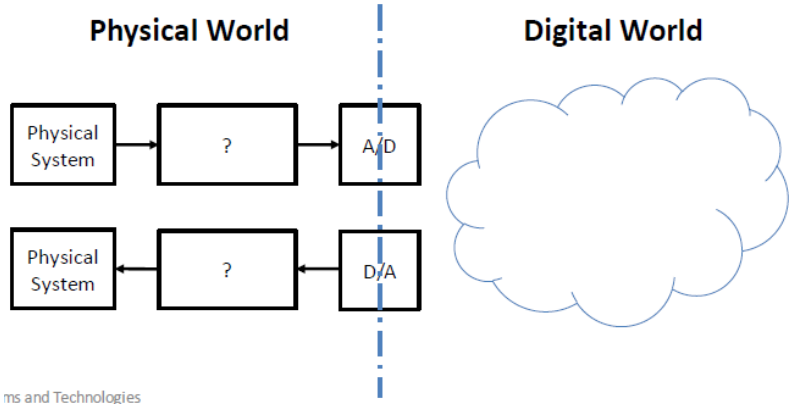
Simone Coco

Cyber-Physical Interface

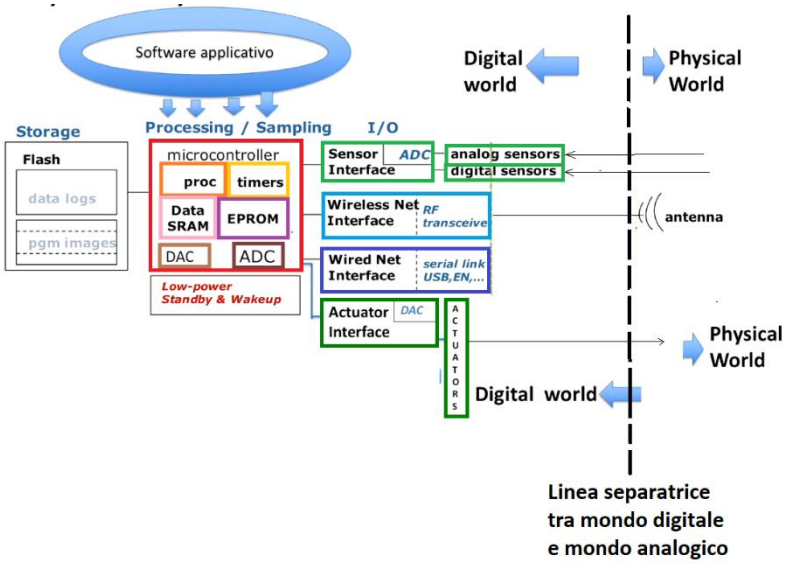
Un sistema IoT può essere definito come sistema Cyber-Physical poiché esso è in grado di interagire (fisicamente) con il mondo reale captando dei dati in input attraverso dei sensori, prendendo delle decisioni con l'uso di un microcontrollore e attuando queste decisioni sul mondo esterno attraverso l'uso di un attuatore.

Interfacciamento tra mondo fisico e mondo digitale

Il mondo reale è **analogico**, mentre i sistemi cyber appartengono ad un mondo **digitale** (infatti vendono anche chiamati device digitali). Quindi il primo problema che si pone è quello dell'iterazione tra due mondi di natura differente: è quindi necessario che i segnali captati dal mondo vengano convertiti da analogico a digitale affinché essi possano essere computati dal microcontrollore e, finita la computazione, è necessaria la conversione da digitale ad analogico per ottenere un segnale che possa essere utile al mondo fisico.



Da questo grafico possiamo ritrovare le componenti principali per il funzionamento di un dispositivo smart:



Il microcontrollore formato dal microprocessore, la memoria statica SRAM, la memoria EPROM (piuttosto che una memoria Flash) e i convertitori DAC e ADC.
Il microcontrollore deve ovviamente consentire l'interfaccia di Input con i sensori: se questi captano dei segnali analogici allora verranno convertiti altrimenti, se sono digitali, non si necessita di alcuna conversione visto.
Deve anche garantire, chiaramente, un'interfaccia di Output per collegarsi agli attuatori che "traducono" le indicazioni del microcontrollore in azioni fisiche da compiere (esempio: un motore pilotato da microcontrollore o un led).
Al di sopra del microcontrollore troviamo il Software applicativo che può stare in parte sul microcontrollore ed in parte su fog o su cloud (non per forza sul nodo).

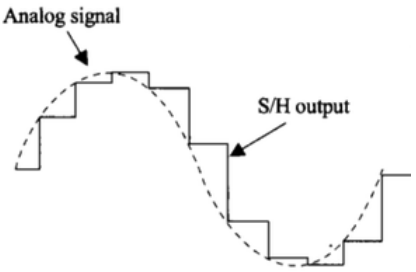
All'interno del nostro nodo possiamo anche notare due interfacce particolari: la Wireless Net Interface che implementa uno dei protocolli studiati in precedenza come l'IEEE 802.15.4 che si connette al ricetrasmettitore e all'antenna e la Wired Net Interface che sfrutta i protocolli per connessioni wired.

Non sempre occorre la conversione analogico/digitale o digitale/analogica perché a volte il sensore e/o l'attuatore possono fornire degli input o degli output che possono essere direttamente digitali.

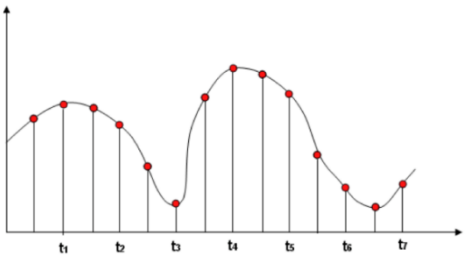
Segnali analogici e teorema di Nyquist-Shannon

Abbiamo studiato vari tipi di sensori, la maggior parte di essi sono dei trasduttori che producono un segnale analogico in input al microcontrollore.
Quando sono nati i primi computer l'unica iterazione con il mondo esterno era quello di interfacciare l'utente (equivalente del mondo esterno) ed il computer e l'unico problema era quello di utilizzare in modo opportuno le schede perforate.

Contemporaneamente però nel campo delle telecomunicazioni si stava passando dalle comunicazioni analogiche a quelle digitali.
Con lo sviluppo delle tecnologie e dei computer si pensò di trasformare le onde analogiche in onde digitali leggibili ai dispositivi digitali.



Quindi ci si pone il seguente problema: **come fare a trasformare un segnale da analogico a digitale?** Idealmente è quello di far si che il segnale digitale approssimi in modo fedele la sinusoide che rappresenta (quindi, idealmente, il segnale "a gradini" che si viene a formare deve essere il più fedele possibile alla sinusoide che rappresenta).
Piuttosto che prendere l'intera sinusoide da "digitalizzare" si prendono dei campioni di quest'ultima: più piccoli sono gli intervalli di tempo tra un campione e l'altro e più si approssima meglio il segnale viene approssimato dai suoi campioni.



Questo è stato quello che pensarono Nyquist e Shannon che dedussero un teorema fondamentale nelle telecomunicazioni per la conversione analogico/digitale, si parla del teorema del campionamento o **teorema di Shannon-Nyquist:**
"Se si campiona un segnale a banda limitata f_{max} con una frequenza f_c tale che $f_c \geq 2f_{max}$ allora non si ha perdita di informazione"

Sfruttando il teorema di Shannon e grazie allo sviluppo tecnologico è stato possibile far interagire il mondo digitale e quello analogico, ha favorito la nascita di sistemi embedded, fino ad arrivare al mondo IoT odierno.
Quando un sensore cattura un segnale analogico, questo deve essere convertito dal convertitore apposito, quindi il convertitore prende in input il segnale e lo converte nel corrispondente digitale o analogico che sia; tuttavia esso non esegue il campionamento. Per campionare il segnale:

1. Si deve garantire che quest'ultimo sia a banda limitata (o non potremmo applicare il teorema di Nyquist-Shannon)
2. Si deve scegliere un periodo di campionamento per prelevare i campioni.

I sensori trasducono il segnale analogico che captano in segnale digitale ma non si occupano del campionamento, quindi è lecito chiedersi: quale parte del nostro schema si occupa di campionare i segnali digitali provenienti dal mondo esterno e captati dai sensori decidendo il tempo di campionamento?

Dovendo acquisire i dati in tempi limitati, quale parte si cura di avere una ricezione dei dati che avvenga entro i tempi stabiliti con il teorema di Nyquist?
Il microcontrollore ha un determinato Throughput, quindi avrà dei limiti di velocità: ha un limite di bit che si possono smaltire in un determinato tempo; se la frequenza di campionamento del segnale che dobbiamo campionare è maggiore del Throughput il microcontrollore non è in grado di utilizzare il sensore senza perdere informazioni.
Dato il throughput massimo possiamo capire quante sorgenti possiamo utilizzare: se ho un throughput massimo di 100Kbyte/s e sorgenti che, campionate, ognuna genererebbe 25Kbyte/s allora potremo gestire solo 4 di queste sorgenti.
Il teorema di Shannon ci serve per capire, se con il microcontrollore a disposizione riusciamo ad evitare perdite di informazioni dei dati in input dai sensori.
O altrimenti, sapendo che un sistema deve gestire un'applicazione, conoscendo il throughput necessario, possiamo capire quale sarebbe il microcontrollore in grado di garantirlo.
Quindi è bene dotarci di metodi e mezzi per quantizzare queste quantità per capire quanto throughput si riesce smaltire e quindi che applicazione si può sviluppare o viceversa: data un'applicazione, è possibile trovare un processore che regga questa applicazione.

Studiamo adesso la f_{max} presa in riferimento per l'applicazione del teorema di Shannon-Nyquist: essa è praticamente metà della frequenza minima affinché il campionamento possa avvenire senza perdita di informazione [Ricorda: $f_c \geq 2f_{max}$]

$$f_{MAX} = \frac{1}{2\left(T_{ACQ} + T_{CONV} + T_{AP}\right)}$$

f_{max} è formata da vari parametri:

- Primo fra tutti è il tempo di acquisizione: idealmente il tempo per prendere un campione è praticamente nullo: viene preso "automaticamente" il segnale sull'opportuno asse di riferimento. Tuttavia, nella realtà, il dispositivo sample-to-hold perde del tempo per "catturare" un campione, introducendo quello che viene definito con il nome di **tempo di acquisizione T_{ACQ}** ; questo tempo introduce una determinata **latenza.**

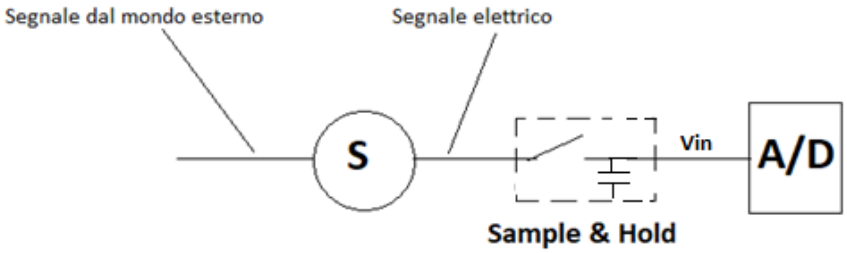
- Una volta che viene acquisito questo valore, esso è ancora un valore analogico: deve quindi essere convertito in segnale digitale; per questo scopo viene incontro il convertitore analogico/digitale. Per questa azione di aggiunge un'ulteriore latenza, stavolta data dal **tempo di conversione T_{CONV}.** Vi è inoltre anche un terzo parametro che per adesso trascuriamo.

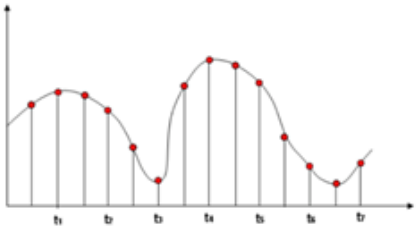
- Un terzo parametro è il **T_{AP}** che indica il massimo **tempo di apertura**, tuttavia, è sufficientemente piccolo per considerarlo approssimabile a zero.

Conoscendo quindi il sistema di acquisizione ed il microcontrollore utilizzato si possono conoscere questi parametri elencati; così facendo si può determinare f_{max} e quindi, di conseguenza, si può stabilire se il sistema regge o meno il segnale (se viene soddisfatto il teorema di Shannon-Nyquist o meno): significa che il sistema riesce a processare il segnale analogico.

Sample & Hold

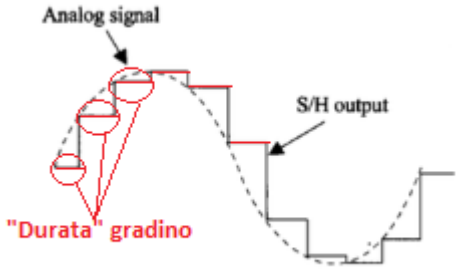
Il campionamento spetta ad un componente che prende il nome di Sample & Hold: esso è il dispositivo che si occupa di trarre dai segnali analogici i campioni che servono per campionarlo e successivamente convertirlo in digitale.



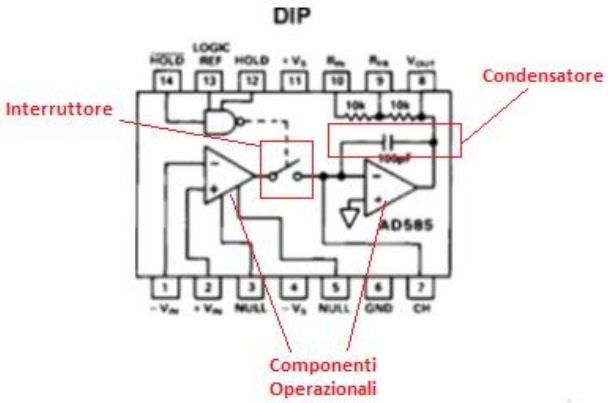


Il Sample & Hold si colloca tra il sensore ed il convertitore analogico/digitale. E' un blocco che si occupa di campionare il segnale, funzionando come un interruttore collegato ad un condensatore: si chiude quando deve campionare (il circuito si chiude e carica il condensatore) e si apre quando non deve prendere alcun campione.

Il condensatore serve per "memorizzare" il valore che viene passato al convertitore. E' importante che venga memorizzato il valore da passare al convertitore poiché quest'ultimo, dopo aver ricevuto un segnale analogico, impiega un determinato tempo T_{CON} per dare in uscita il corrispondente segnale digitale; quindi è importante che venga mantenuto costante il valore Vin caratterizzante il campione affinché questo possa essere convertito. [Questo nel grafico si traduce come "la durata del gradino": il segnale resta in ingresso per tutta questa durata].
[**Sample: campionare & Hold: manentere**]

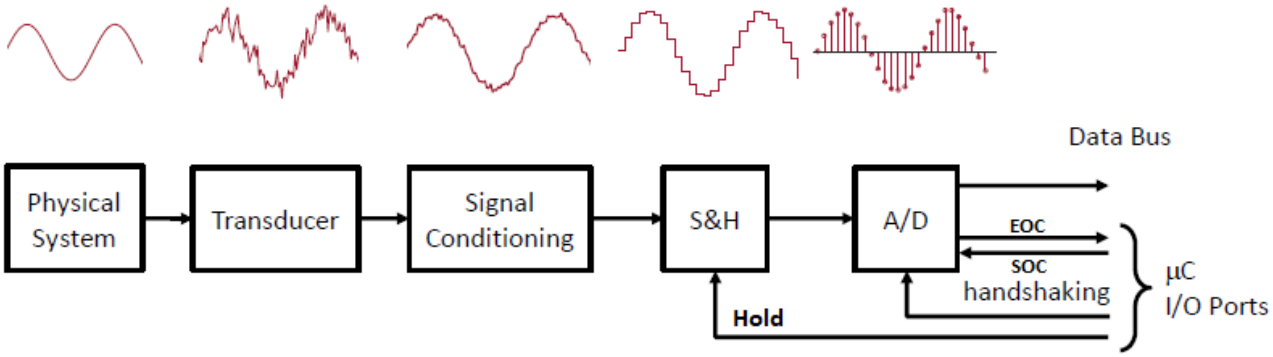


Lo schema che abbiamo visto è molto semplicistico poiché i condensatori tendono a scaricarsi verso le vie che trovano a disposizione. Servono quindi altri schemi utili e realistici.



Vi è un interruttore che viene pilotato dal segnale sul pin "hold". Vi sono due apmlificatori operazionali che servono per evitare che il condensatore che si possa scaricare verso la massa o verso all'interruttore

Catena di acquisizione di un dato ad unico canale



- Partendo da un sistema fisico, dobbiamo misurare delle grandezze attraverso un sensore (un trasduttore): esso trasduce il segnale di una certa natura in una tensione [analogica].
- Il segnale passa ad un componente che prende il nome di Signal Conditioning che si comporta come filtro per ricostruire l'insieme di tensioni uscenti dal trasduttore e, in alcuni casi amplifica o attenua il segnale.
- Dopo di che vi è il Sample&Hold che attraverso un segnale che chiameremo Hold effettua il campionamento: il segnale Hold pilota un interruttore che si chiude e/o si apre a seconda del suo valore: quando il segnale vale 0 l'interruttore è aperto e il Sample and Hold assume lo stato di Hold, quando il segnale vale 1, l'interruttore viene chiuso e lo stato passa a Sample.
- A questo punto il segnale passa al convertitore analogico/digitale (A/D). Un convertitore A/D possibile è quello ad approssimazioni successive.
- Il convertitore è comandato da alcuni segnali:
  Quando il convertitore ha terminato ed ha in uscita il risultato digitale, esso lo manda al Data Bus (che manda i dati verso un dispositivo o verso la CPU).
  Come si può notare vi sono due line opposte nel convertitore A/D: chiameremo queste EOC: End Of Conversione ed SOC: Start Of Conversion.
  Questi due segnali sono praticamente comandi che dicono quando iniziare e terminare la conversione, rispettando il tempo di conversione $T_{CONV.}$ Il convertitore indica di aver terminato la conversione attraverso il segnale EOC, così da poter passare il segnale al device che lo richiede (sia esso un qualunque dispositivo o la CPU stessa).

  Affinché tutto possa funzionare è necessario un **timing:** si deve generare il segnale di Hold per comunicare al Sample and Hold di aprirsi e solo dopo si può comunicare il segnale SOC (start of conversion), dopo di che, una volta finita la conversione, si trasmette il segnale EOC (end of conversion) che segna la terminazione della conversione. Infine, deve essere mandato il segnale nel Data Bus, una volta eseguite queste azioni il segnale di Hold passa a 0 ed il convertitore è pronto a convertire un altro segnale; in sequenza, devono essere eseguite le seguenti azioni:

  1. Comunicazione di chiudere il circuito attraverso il segnale Hold
  2. Trasmissione del segnale SOC (Start Of Conversion)
  3. Trasmissione del segnale EOC (End Of Conversion) dopo che è passato T_{CONV}
  4. Trasmissione del segnale nel Data Bus
  5. Comunicazione di aprire il circuito attraverso il segnale Hold = 0

  Queste quattro operazioni devono essere sincronizzate con un'operazione di **timing**: i tempi per queste 4 azioni sono dettate dal **microcontrollore**, che attraverso il **timer** (che è programmabile via software) si possono instaurare una serie di timeout per definire le deadline per ogni operazione. Chiaramente, dovremo rispettare il timing rispettando il teorema di Shannon-Nyquist.

Rianalizzando la formula allora possiamo dire che:

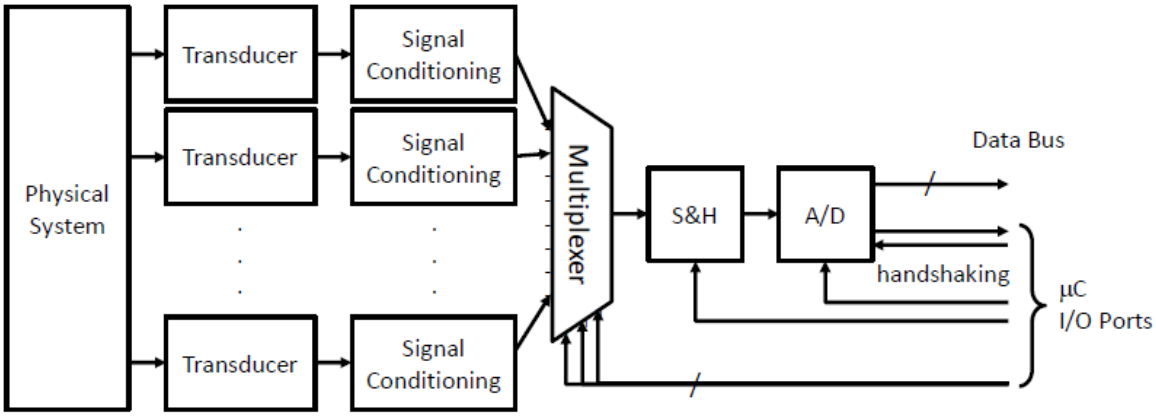
- il T_{ACQ} è **il ritardo imposto dal sample & hold**
- il T_{CONV} è il **tempo di latenza del convertitore per la conversione**
- il T_{AP} è il **ritardo introdotto dal'interruttore** poiché anch'esso richiede del tempo per aprirsi e/o chiudersi, tuttavia, questo è un tempo molto piccolo che viene spesso trascurato.

$$f_{MAX} = \frac{1}{2\left(T_{ACQ} + T_{CONV} + T_{AP}\right)}$$

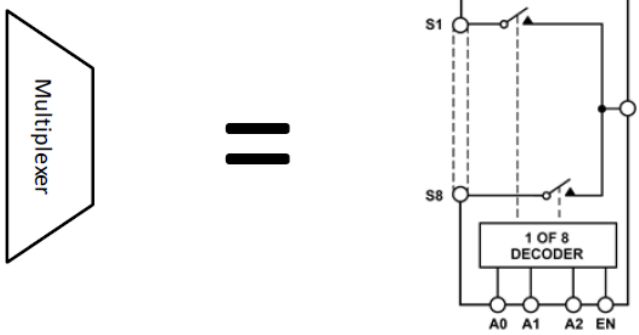
Quindi, riguardando la formula, possiamo dire che la somma dei tre tempi impiegati (quindi la somma dei tre tempi dettati dal timer) deve essere pari a T_{MAX} con quest'ultimo pari all'inverso di f_{MAX} così facendo otteniamo la frequenza che ci servirà per applicare il teorema di Shannon-Nyquist, ricordando che $f_c \geq 2\, f_{MAX}$.

Catena di acquisizione di un dato a canale multiplo

Normalmente, in uno smart objects vi sono collegati più sensori, quindi teoricamente servirebbero più canali di conversione, con più Sample & Hold, più convertitori, etc.. (più in generale: servirebbero più schemi simili a quelli riportati).
In realtà sarebbe oneroso e scomodo utilizzare così tanti dispositivi per instaurare più schemi, per questo motivo si instaura uno schema del tipo:



Da un sistema fisico (che intendiamo come mondo reale) possono esservi collegati tanti sensori quanti sono i valori che si vogliono misurate (ad esempio, uno per la temperatura, una per la pressione, uno per l'umidità, etc…).
Dai sensori i segnali passano ai Signal Conditioning: avremo tanti Signal Conditioning quanti sono i sensori, dunque quante sono le grandezza da misurare.
Dopo di che l'uscita di ogni Signal Conditioning passa ad un **multiplexer analogico,** esso equivale ad un componente che può essere visto come rappresentato da un dispositivo avente un insieme di ingressi e di interruttori. Esso sulla base di segnali di comando in ingresso (sui pin A0, ..An), può decidere di chiudere l'interruttore per scegliere quale degli ingressi portare in uscita chiudendo il corrispettivo interruttore, esempio: s1 per A1. Il decoder associa una combinazione ad ogni segnale, in questo modo può mappare ogni interruttore e chiudere quello corrispondente al segnale analogico di comando in ingresso.



Il multiplexer può avere diversi pin di controllo, dal numero di essi dipende il numero di bit di controllo: per 8 interruttori si utilizzano 3 bit in quanto $2^3 = 8$.
Il multiplexer "permette" la sola presenza di un segnale alla volta, quindi in un determinato istante, lo schema di riduce ad essere quello per comunicazione a canale unico, visto prima.
Il multiplexer ha però bisogno di conoscere quale pin è abilitato alla conversione per chiudere l'interruttore interessato e convertire l'informazione esatta: quindi dal microcontrollore, oltre ai collegamenti già visti, vi sono anche quelli per i pin di controllo.

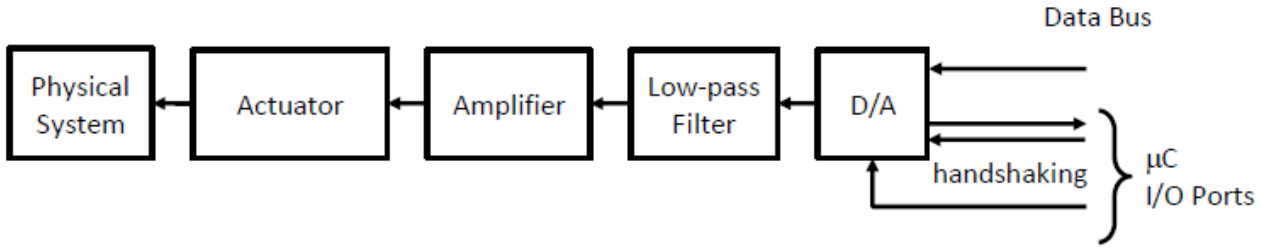
Chiariamo adesso come avviene il timing nel caso di multicanale: supponiamo inizialmente che tutti i segnali provenienti dal mondo esterno possano essere campionati ad una stessa frequenza f_c , supponendo quindi che abbiano tuti la stessa banda.
Ci chiediamo allora coma varia il tempo di conversione rispetto al caso in cui si aveva un'unica linea di acquisizione (canale unico);
Utilizzando N canali avremo che la conversione deve avvenire N volte più veloce, quindi il tempo di conversione deve essere 1/N volte più piccolo del caso di canale singolo:

$$T_{CONV} = \frac{1}{N} T'_{CONV}$$

Con T_{CONV} pari al tempo di conversione nel caso multicanale e T'_{CONV} al tempo di conversione nel caso si canale unico
Quindi nel caso di multicanale si necessita di un **Sample & Hold ed un Convertitore A/D più performanti** per non sforare l'intervallo di campionamento.
Si deve far in modo che ogni sensore venga campionato come se gli altri non vi fossero, con un tempo tanto più piccolo quanti sono i sensori utilizzati.

Output verso il sistema fisico

Per quanto riguarda l'uscita verso il mondo fisico abbiamo una situazione duale:



Attraverso le porte, il microcontrollore passa i dati (digitali) ad un convertitore Digitale/Analogico, successivamente, il segnale (che è stato convertito in analogico) passa da un filtro (solitamente un filtro passa basso); attraverso il filtro di genera un segnale adatto a pilotare un amplificatore; l'amplificatore è collegato all'attuatore che convertirà il segnale elettrico ricevuto in segnale utile al mondo esterno.

Output verso il sistema fisico

Per approcciarci alla progettazione di un sistema con un adeguato numero di sensori (in modo tale che non vi siano problemi per quanto riguarda il throughput ed altri parametri), iniziamo ad introdurre un modello di interazione:

1. Chi genera i comandi per l'acquisizione di segnali analogici? Il dispositivo che genera i comandi è chiaramente il microcontrollore che "pilota" i sensori, attraverso le porte e il timer
2. Come viene gestito il timing? Il timing viene gestito dal timer del microcontrollore che stabilisce dei tempi di deadline per le operazioni di conversione
3. Chi prende iniziativa? Il segnale EOC annuncia che l'informazione è pronta per essere letta. Vi possono essere due possibilità: seguendo una tecnica di "polling" osservando continuamente quando il segnale EOC assume valore true (segnala la presenza di dati convertiti); altrimenti si può sfruttare un modello con interrupts: quando un dato è pronto viene inviato un interrupt che interrompe la CPU per richiede di essere processato. E' importante scegliere una tecnica adatta perché da essa dipende il throughput massimo smaltibile e quindi ciò ci dice quanti sensori si possono utilizzare.

Gestione dell' Input/Output

Ci poniamo il problema di connettere il microcontrollore con il mondo esterno. Abbiamo specificato con il teorema di Nyquist Shannon che il campionamento dei dati deve avvenire con una frequenza pari a:

$$f_c \geq 2f_{MAX} \implies T_c = \frac{1}{f_c} \leq \frac{1}{2\,f_{MAX}}$$

Supponiamo di avere un qualsiasi sistema di I/O: esso sarà in grado di scambiare un certo numero di byte al secondo, questa quantità di chiama **throughput**, lo chiameremo $f_{I/O}$ Dal teorema di Nyquist-Shannon, sappiamo che per smaltire un determinato throughput (un determinato numero di byte al secondo) necessiteremo della seguente condizione:

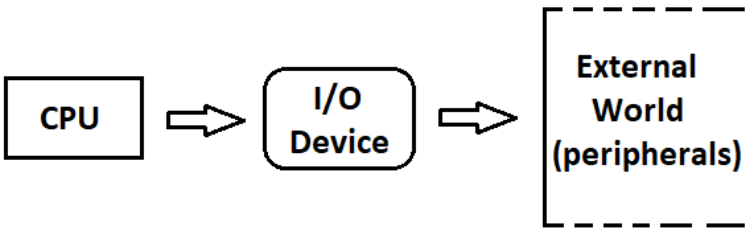
$$f_{I/O} = \frac{1}{T_{I/O}} \geq f_c = 2\,f_{MAX}$$

In questo modo stiamo mettendo in relazione le caratteristiche I/O del sistema con il mondo fisico: mettendo in relazione la massima quantità di dati che si può smaltire (throughput) con la massima frequenza del segnale che i sensori possono acquisire.

Se ho un sistema che introduce una certa latenza per smaltire un certo numero di byte pari a $T_{I/O}$, e quindi caratterizzato da un certo throughput, la massima frequenza del segnale percepito dal sensore è pari a f_{MAX} :

$$f_{MAX} \leq \frac{1}{2}\,f_{I/O} = \frac{1}{2T_{I/O}}$$

Un sistema di I/O mette in comunicazione il mondo interno (il microcontrollore) con il mondo esterno, attraverso la periferica. Questo è possibile mediante delle apposite interfacce: gli I/O device (adattatori).



Gli I/O device devono adattare:

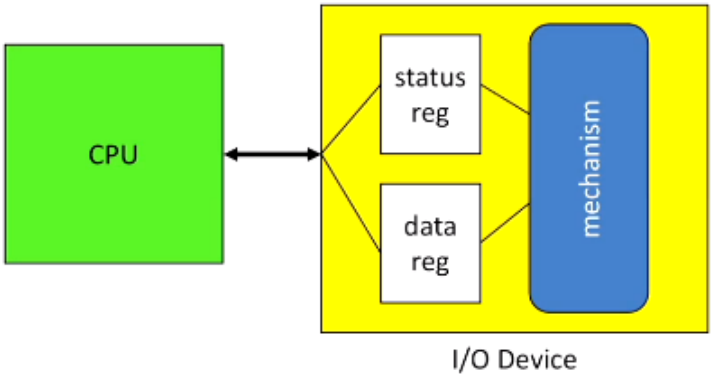
1. Velocità e rappresentazione dell'informazione
2. Protocollo di comunicazione (meccanismi di sincronizzazione)
3. Livelli e tipi di segnale

Le interazioni che si devono gestire mediante modelli di interazione e sincronizzazione allora riguardando la CPU con l'I/O device e successivamente l'I/O device con il mondo esterno; la CPU, infatti, non interagisce mai direttamente con le periferiche.

- **Interazione CPU - I/O Device**: la CPU potrebbe condividere il bus con l'I/O device; essa comunica con l'I/O device come se questo fosse una memoria: vi accede e può scrivere o leggere su/da esso. Viene utilizzato lo status register per la sincronizzazione ed il data register per lo scambio dei dati.
- **Interazione I/O - Periferiche**: il collegamento può avvenire, ad esempio, con una seriale (UART). La sincronizzazione spetta a dei meccanismi di handshaking.

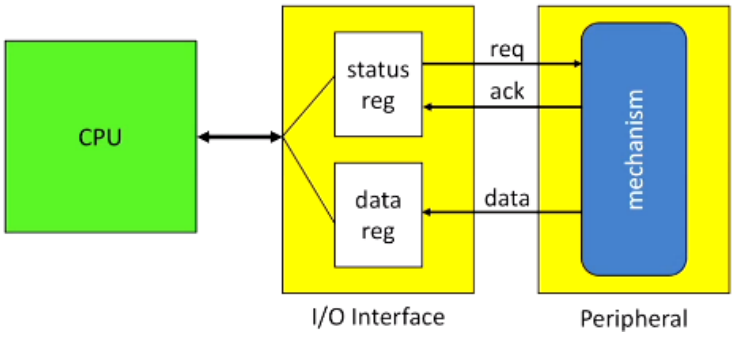
Interfaccia CPU <=> I/O Devices

La CPU viene collegata mediante bus all'I/O device (al bus possono anche essere collegati altri dispositivi come le memorie).
Esempio: la CPU deve comunicare un byte, allora, mediante il bus, lo scrive nel data register dell'I/O Device. Se si devono trasmettere più byte è bene che questi vengano smaltiti alla giusta velocità perché se l'I/O Device non "preleva" il dato nel data register in tempo, la CPU potrebbe sovrascrivere il valore con un altro byte. Quindi serve un meccanismo di sincronizzazione, per avvisare alla CPU che l'I/O device è pronto a ricevere un altro dato.
Per questo viene utilizzato lo status register: esso contiene delle informazioni che la CPU può leggere per vedere se il data register è libero o meno. Quindi per la sincronizzazione la CPU entra nello status register, legge se può scrivere e qualora potrà farlo allora scriverà nel Data Register. Per comunicare lo stato di libero/occupato lo status register setta un flag. Se il flag è settato la CPU capisce che il data register è occupato dal byte precedente e quindi attenderà



Interfaccia I/O Devices <=> Periferiche

Una volta che i dati provenienti dalla CPU vengono acquisiti dall'I/O Device, questi vengono "inoltrati" alla periferica: l'I/O device fa quindi da tramite tra la CPU e la periferica.
Anche in questo caso, la periferica legge o scrive i dati all'interno del Data Register. Chiaramente la periferica dovrà aspettare che la CPU, qualora dovesse leggere (e non scrivere) i dati, svuoti il registro Data e quando ciò avviene la periferica deve essere "avvisata", si necessita dunque di una sincronizzazione anche in questo caso.

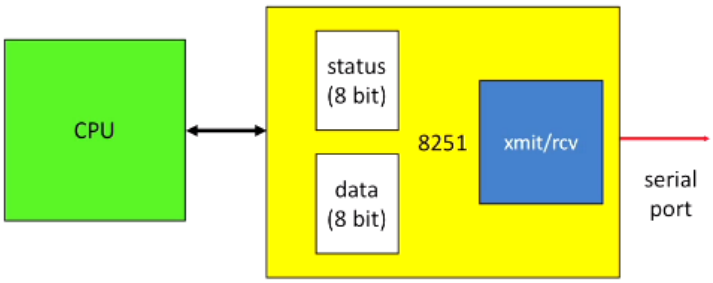


La sincronizzazione avviene attraverso i segnali di request e acknowledgement con due possibili metodi: **strobe protocol** o **handshake protocol (o handshaking)**.

Analizziamo l'handshake protocol:
L'**handshaek protocol** è un protocollo per sincronizzare un **master** ed uno **slave**.

1. Il master avvia una richiesta di ricevere i dati. Il client riceve questa richiesta e fronte di essa manda un ACK al master.
2. Una volta che l'ACK è stato inviato si comunica al master che la periferica è pronta al trasferimento dati, sia in input che in output.
3. Quindi il master invia i dati verso la periferica e li tiene lì per un certo tempo: fintanto che la request resta attiva (settata ad 1);
4. quando viene esaurito questo tempo, la request viene settata a 0, viene disattivata e cessa lo scambio di dati (chiaramente la comunicazione può avvenire anche dal client al server purché sia settato il segnale di request).
5. Una volta cessata la request lo slave si "disconnette" settando l'ACK a 0.

Abbiamo trattato l'UART come interfaccia seriale per connettere un device esterno con il microcontrollore.
L'UART utilizza la **8251 CPU Interface**:



Per interfacciarsi con la CPU, l'UART al suo interno opera con i registri di data e status;
il registro di stato in input quando i dati sono stati letti dal registro dati avvisa la periferica, in output l'avviso del registro di stato viene inviato quando i dati sono stati scritti sul data register.
L'indirizzo di stato e quello dei dati devono essere indirizzati (vengono trattati come memorie quando essi vengono letti o scritti): alcuni processori hanno delle istruzioni di base, attraverso l'I/O Mapping e la CPU per scrivere o leggere deve utilizzare una particolare istruzione. Se non è prevista questa modalità la CPU utilizza un Memory Mapping in cui eseguirà le operazioni con l'inidirizzamento, si dice che eseguirà una store.

Problemi di performance dell'I/O

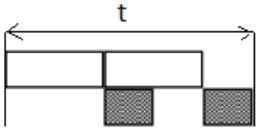
Le performance dell'I/O migliorano, negli anni, molto più lentamente rispetto a quelle delle CPU.

Sfruttiamo la **legge di Amdahl** per capire meglio lo scenario:
supponendo di avere un sistema in cui il 10% del tempo per eseguire un programma viene speso per eseguire istruzioni di I/O e il 90% rimanente con istruzioni che non comportano operazioni di I/O; se in un sistema con queste caratteristiche 10 volte maggiori otteniamo una performance che è soltanto 5 volte maggiore: si ha una perdita del 50% di performance.
Se si prende una CPU 100 volte più veloce si ottiene una performance 10 volte migliore: si ha una perdita di performance del 90%.
Quindi possiamo dire che le operazioni di I/O impattano negativamente con le performance, diremo che **gli I/O device formano un collo di bottiglia (bottleneck).**
È bene dunque preoccuparsi delle performance dell'I/O; un indice importante per le performance di I/O è la percentuale di CPU Time, ovvero la percentuale del tempo che la CPU spende per operazioni di I/O.
Durante un'operazione il tempo speso per un intero lavoro può essere visto come sommatoria tra il tempo di CPU ed il tempo speso nelle istruzioni di I/O:

$$\text{Time(workload)} = \text{Time(CPU)} + \text{Time(I/O)}$$

In figura, il CPU Time è quello in bianco, l'I/O time in grigio.
Una possibile ottimizzazione sarebbe l'esecuzione in parallelo delle operazioni di CPU che non richiedono I/O e le operazioni che invece lo richiedono. Così facendo di perde meno tempo, quindi dalla formula scritta precedentemente può essere sottratta una certa quantità temporale data dalla sovrapposizione dei tempi di CPU e di I/O.

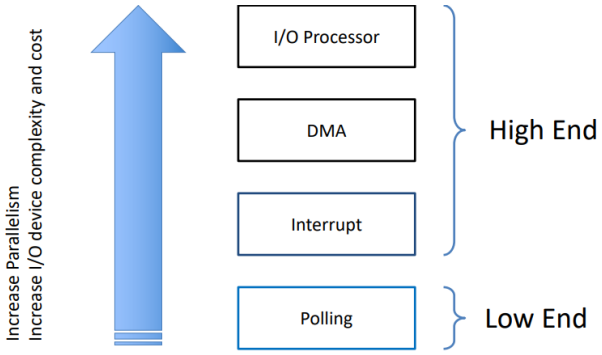
$$\text{Time(workload)} = \text{Time(CPU)} + \text{Time(I/O)} - \text{Time(Overlap)}$$



Con il time di overlap che indica il tempo in cui il blocco bianco e quello grigio si sovrappongono.
Esso rappresenta il **grado di parallelismo tra la CPU e i device di I/O.**

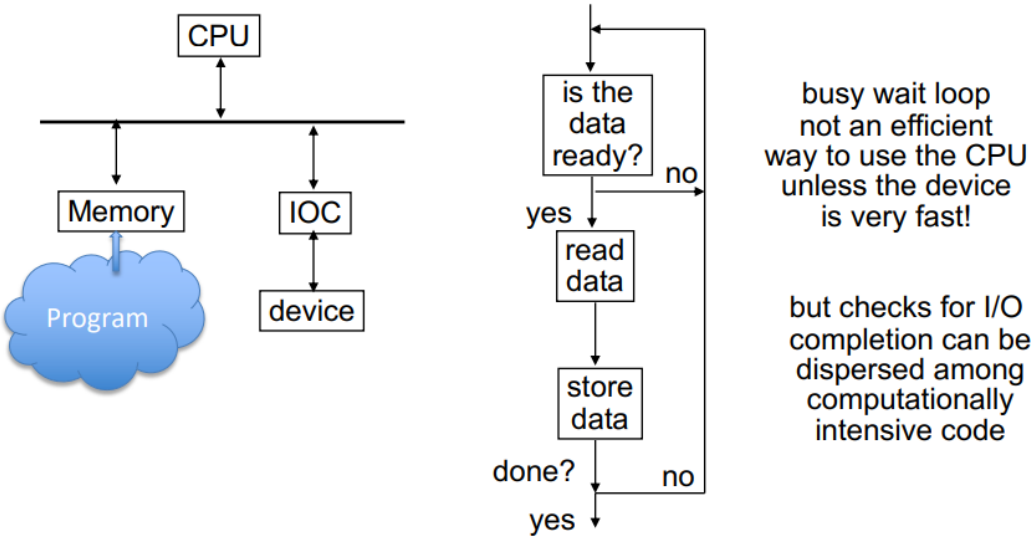
Possiamo classificare le **tecniche di gestione dell'I/O** a seconda del **grado di parallelismo:**

0. **Polling:** Ha un parallelismo nullo (non prevede parallelismo)
1. **Interrupt:** Ha un medio livello di parallelismo
2. **DMA:** Ha un alto livello di parallelismo
3. **I/O processor:** Ha un altissimo livello di parallelismo



Tecniche di gestione dell'I/O: Polling

Il polling ha un basso grado di parallelismo.
La CPU osserva lo stato del registro data, se il registro conferma la possibilità di poter scrivere/leggere allora la CPU esegue, altrimenti attende e riprova in un secondo momento, continuando ad interrogare il registro di stato.



Il polling coinvolge al 100% la CPU: essa deve occuparsi di interrogare il registro di stato del device (deve effettuare il polling). Se il numero di device aumenta si ha una serie di interrogazioni che fanno perdere tempo.
Quindi con questa tecnica viene perso molto tempo: anche se la CPU è molto veloce i device di I/O sono molto più lenti, quindi con questa tecnica è molto probabile che la CPU resti più volte ad attendere che le risorse siano disponibile, quindi è molto probabile che essa resti in attesa di sincronizzarsi.
L'**efficienza del polling** è dato da:

$$\eta = \frac{T_{TRANSFER}}{(N_{DATA\ IS\ NOT\ READY} + 1)T_{CHECK} + T_{TRANSFER}}$$

Dove:

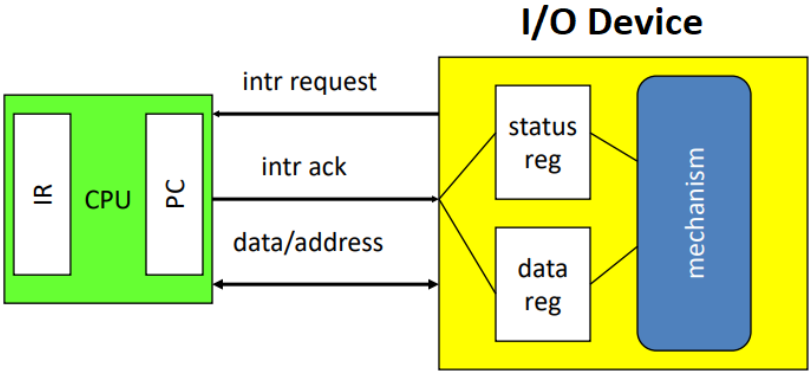
- $T_{TRANSFER}$**:** tempo impiegato per il trasferimento
- $N_{DATA\ IS\ NOT\ READY}$**:** numero di volte che il dato non è pronto [$N_{DATA\ IS\ NOT\ READY} + 1$ indica il numero di volte che il dato non è pronto + la volta in cui è pronto]
- T_{CHECK}**:** tempo sprecato a domandare se la risorsa è pronta

L'efficienza è data dal rapporto tra il tempo totale per il trasferimento e tutto il tempo sprecato in cui la CPU è rimasta in stato di attesa

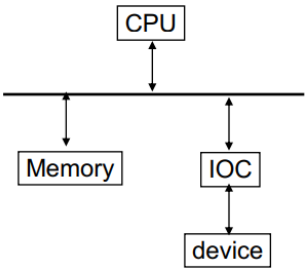
Tecniche di gestione dell'I/O: Interrupt I/O

Subito sopra alla tecnica di polling vi è quella dell'Interrupt. Il polling è una tecnica molto svantaggiosa perché l'assenza totale di parallelismo porta ad una perdita di tempo eccessiva.

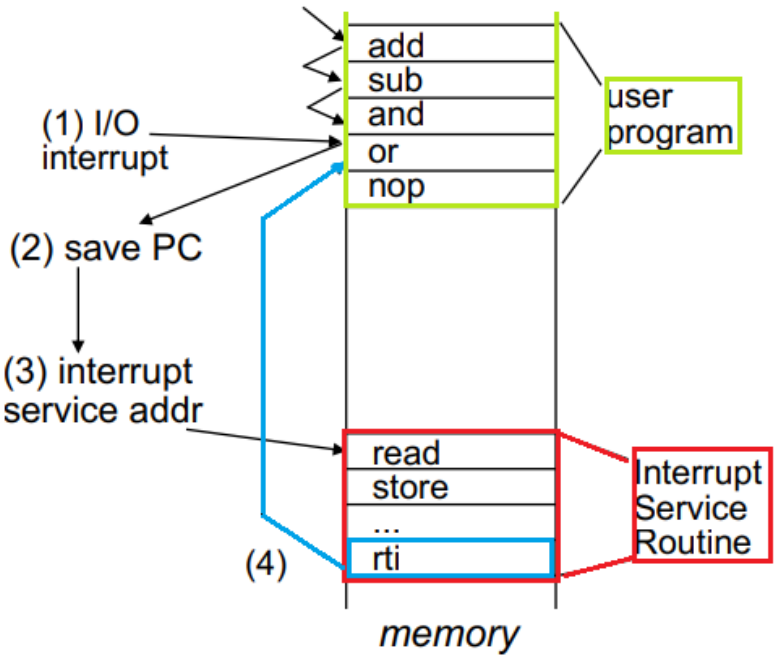
L'idea della tecnica ad interrupt è che **il dispositivo va ad interrompere la CPU durante l'esecuzione, informandola che il dato è pronto,** "forzandola" a leggere/scrivere il dato piuttosto che continuare ad eseguire il compito che stava eseguendo prima dell'interrupt.



In questo funzionamento si vanno ad aggiungere due linee di trasmissione: **interrupt request (intr request)** ed **interrupt acknowledgement (intr ACK)** per stabilire il funzionamento di interruzione della CPU: il device invia la richiesta di interrupt e la CPU risponde con un ACK corrispondente. Si ha poi la linea ("classica") per lo scambio di dati ed indirizzi.

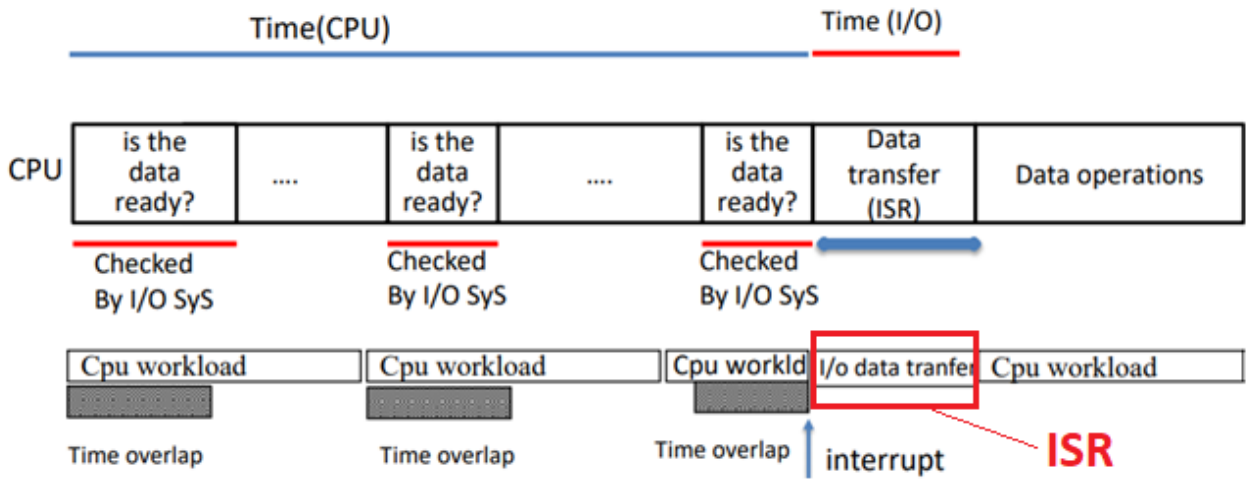


Supponiamo che la CPU esegue un user program; ad un certo punto avviene un interrupt: riceve una **interrupt request** che interrompe l'esecuzione del programma e "forza" la CPU a leggere o scrivere da/sul Data Register. La CPU allora interromperà l'esecuzione del programma salvandone lo "scenario" nel program counter (PC): salvando la prossima istruzione ed effettuerà le operazioni sul I/O Device.
Le operazioni che devono essere effettuate sul I/O Device si trovano scritte in una funzione specifica: L'**Interrupt Service Routine (ISR)** o **Handler**. Quindi una volta che la CPU salva lo stato nel PC in seguito all'interruzione del programma, viene avviata l'ISR per gestire l'I/O Device. L'ultima istruzione dell'ISR è la **RTI**, l'istruzione di "Return from Interrupt" che comunica alla CPU che l'interrupt è finita e che quindi la CPU dovrà ricaricare lo scenario dal program counter (caricando quindi la prossima istruzione su cui è stato interrotto) e dovrà procedere con le istruzione in modo trasparente, come se nulla fosse accaduto. Il programma che è estato interrotto prende il nome di **foreground program**.
La CPU comunica la propria disponibilità a gestire l'interrupt grazie ad un ACK che viene comunicato all'I/O Device in risposta alla interrupt request.



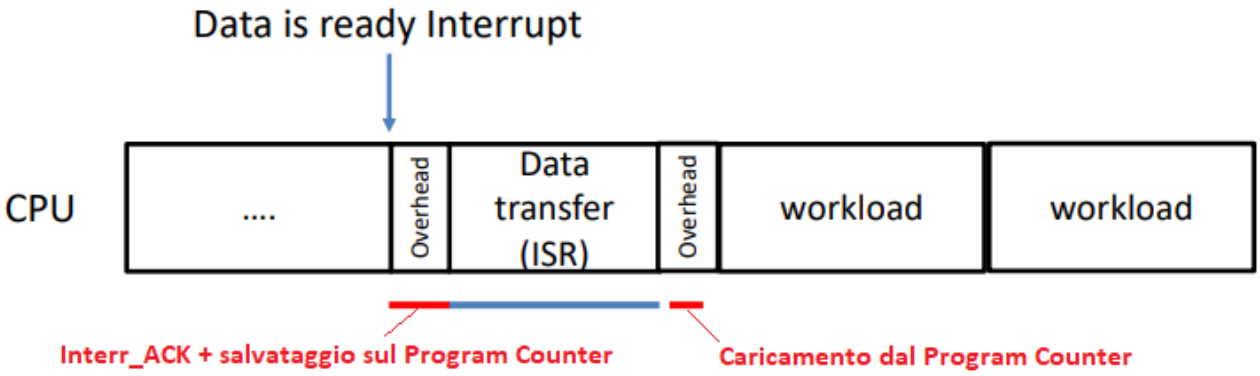
Calcoliamo allora in caso di interrupt l'efficienza, così come fatto per il polling.

Come si può notare la CPU sta lavorando e il check dello stato non spetta a lei ma all' **I/O system** è il device che "perde tempo" nell'interrogazione continua sullo stato del dato (se è pronto o meno) e questo tempo viene perso in parallelo alla CPU, non intaccando i tempi di CPU.
E' come se il device fosse "autosufficiente" per interrogare continuamente il registro di stato, quando lo stato segna la presenza di un dato viene inviato l'interrupt.



Nel caso del polling, invece, la richiesta veniva fatta continuamente dalla CPU e questo intaccava nelle prestazioni; ciò avveniva perché non si aveva il parallelismo ottenuto con l'introduzione dell'interrupt.

Tutte le volte che la CPU deve salvare lo stato nel program Counter a seguito di un interrupt, e che deve poi ricaricarlo quando l'esecuzione dell'interrupt service routine termina (in seguito all'istruzione ISR) si ha un **overhead:** viene perso del tempo per eseguire queste azioni.
All'overhead del salvataggio sul program counter (quindi prima che venga gestito l'interrupt) si somma anche il tempo perso a trasmettere l'ACK di corretta ricezione dell'interrupt da parte della CPU verso l'I/O Device.



Calcoliamo allora come segue l'**efficienza della tecnica con Interrupt**

$$\eta = \frac{T_{ISR}}{T_{OVERHEAD} + T_{ISR}}$$

Dove:

- **T_{ISR}:** tempo impiegato per l'interrupt service routine]
- **$T_{OVERHEAD}$:** tempo sprecato per le operazioni di scrittura/caricamento da/sul program counter + ACK per la conferma di ricezione della richiesta di interrupt.

La CPU in caso di Interrupt spreca il tempo di I/O solo quando viene interrotta: una parte di overhead (ACK + lettura/scrittura program counter) e una parte per eseguire la Internet Service Routine (ISR)

Nel caso di Polling spende il tempo sia per il data transfer che per il check del data ready flag (il flag che viene settato sul state register e quest'ultimo passaggio è ciò che rende più onerosa la tecnica del polling, infatti **non è presente il controllo periodico sulla disponibilità della risorsa nella tecnica con interrupt.**

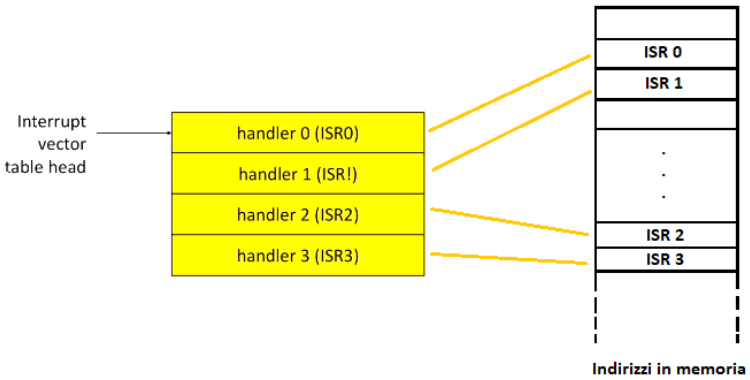
Indirizzamento all'ISR nella tecnica con interrupt

Avendo parlato del meccanismo basato su interrupts, è lecito chiedersi: come fa il microcontrollore ad eseguire il codice dell'ISR? In altre parole, come fa a ricavare l'indirizzo dell'interrupt service routine? Vi sono diverse tecniche:

- **Fixed interrupt:** il microprocpressore si riferisce sempre ad uno stesso indirizzo riservato agli interrupt, a quest'ultimo corrisponderà l'ISR. Dunque, una volta che viene interrotta, la CPU si riferirà a questo dato indirizzo. Questo schema è raramente utilizzato, il suo impiego è previsto per sistemi molto semplici.

- **Vectored interrpt (interrupt vettorizzato):** il device I/O genera la richiesta di interrupt, la CPU manda l'ACK e il dispositivo stesso trasmette alla CPU un vettore, attraverso il data bus, contenente la locazione in memoria della sua ISR.
  Questa soluzione **non è molto flessibile:** una volta che l'interrupt service routine occupa un determinato indirizzo in memoria, il sistema operativo non potrà spostare quest'ultima come meglio crede; la locazione in memoria rimane "bloccata", memorizzando l'ISR, che non potrà essere spostata perché altrimenti dovrebbe essere riprogrammato il device I/O.

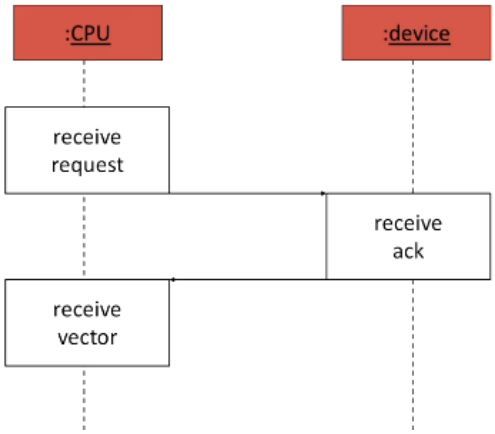
- **Interrupt address table:** è una soluzione che nasce dal compromesso tra la soluzione Fixed Interrupt e la Vectored interrupt;
  una volta che il device avvia una richiesta, la CPU stoppa il processo che stava eseguendo salvandone lo stato nel program counter, dopo di che il device passerà alla CPU un **interrupt vector table head** che sarebbe l'indice di una tabella in cui sono memorizzati gli indirizzi delle interrupt service routine (handler) [I puntatori alle celle di memoria in cui è salvata la ISR].



  Quindi il Device invia attraverso un vettore apposito l'indice della tabella contenente tutti gli indirizzi delle ISR, in questo modo la CPU, facendo riferimento alla tabella, riesce a individuare la locazione dell'ISR di cui necessita.
  Questa soluzione è più flessibile perché se il sistema operativo deve spostare l'indirizzo in memoria dovrà solo cambiare la corrispondente riga nella tabella degli indirizzi, non si necessita quindi di riprogrammare l'I/O device (a differenza della Vectored Interrupt).

Ricapitolando una gestione dell'I/O con tecnica basata sugli interrupt e con vector table head per l'indirizzamento all'ISR funziona come segue:

1. Il Device invia una Richiesta di Interrupt.
2. La CPU riceve la richiesta, stoppa il processo in esecuzione ed invia un ACK
3. Il Device allora invia l'interrupt address table
4. La CPU effettua la chiamata all'handler (ISR) corrispondente, prendendo il suo indirizzo dalla tabella.
5. Il codice dell'ISR esegue le operazioni richieste e cede il controllo alla CPU non appena termina queste azioni attraverso l'istruzione RTI (Return from Interrupt)
6. La CPU ripristina lo stato e passa al foreground program (il programma che stava eseguendo)



La gestione dell'I/O con interrupt alleggerisce il carico sulla CPU, spostandolo sugli I/O device, rendendoli più "intelligenti" e capaci di eseguire alcune azioni che semplificano il lavoro della CPU (rendendo quindi la gestione più efficiente del polling).

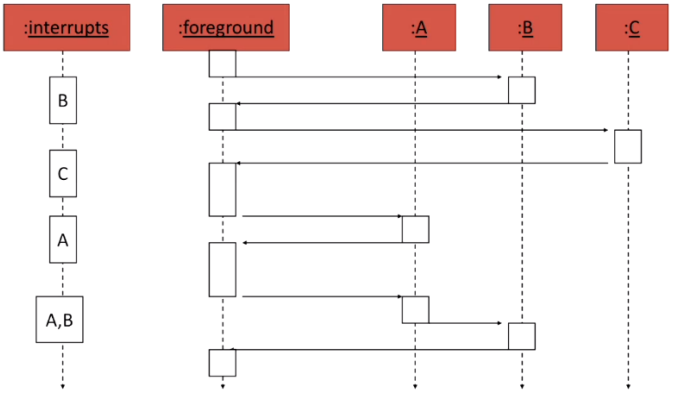
Priorità degli interrupt

In uno scenario in cui vi sia un microcontrollore connesso a più device con tecnica ad interrupts, può capitare che due device possano inviare una richiesta di interrupt contemporaneamente o comunque in un tempo molto breve.
Per questo motivo nasce la necessità di dare una **priorità** agli interrupt, così che la CPU possa sempre capire quale richiesta processare prima e quale dopo. Assegnando la priorità ad una richiesta di interrupt, la CPU eseguirà per prima le ISR che hanno maggior priorità.
Vi sono molte tecniche per l'assegnazione delle priorità.

È bene chiarire che la priorità non si allaccia al concetto di vettore passato dal device alla CPU: il vettore permette di identificare quale ISR eseguire per rispondere ad un determinato interrupt. Non dice nulla sulla priorità: la priorità potrebbe rendere la richiesta più "specifica" ma il concetto di priorità non è correlato al vettore inviato.

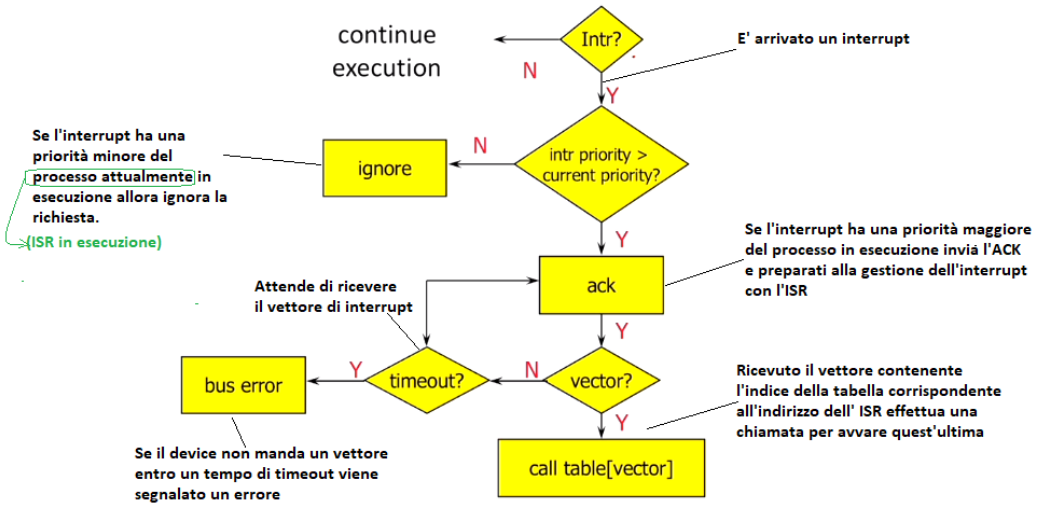
Spesso alla CPU, attraverso software, può essere richiesto di **disabilitare un interrupt**: questo può essere utile quando si hanno delle porzioni di codice che non possono essere interrotte, ad esempio, perché si potrebbe rischiare di modificare uno stato che potrebbe recare problemi; in queste situazioni il software potrebbe disattivare gli interrupt così che la CPU non venga interrotta. In questo caso allora si dice che si effettua un **masking.**

Masking: quando vi sono degli interrupt con una priorità più bassa di quella del processo in esecuzione, essi non vengono presi in considerazione: la priorità non viene presa in considerazione



Parliamo invece di **interrupt non-maskable (NMI)** quando vi sono degli eventi "estremi" che richiedono comunque un'interruzione della CPU, anche quando sta eseguendo una parte di codice "sensibile", è il caso, ad esempio, dello spegnimento della CPU in cui esso dovrà salvare lo stato.

A questo punto possiamo osservare un flow chart che spiega il funzionamento della tecnica di gestione I/O con interrupts con le varie priorità:



Quando utilizziamo un metodo basato su interrupt dobbiamo pagare dei tempi che dipendono da:

- × Esecuzione dell'handler (ISR)
- × Meccanismi di overhead: ACK, priority
- × Salvataggio/Caricamento del programma sospeso dal registro program counter
- × Penalità sull'utilizzo di pipeline (non trattate in questo contesto)
- × Penalità sull'utilizzo di caching (non trattate in questo contesto)

La gestione di I/O con l'interrupt migliora di molto le prestazioni rispetto al polling ma l'evoluzione dei sistemi hanno introdotto l'esigenza di introdurre dei meccanismi più performanti di quest'ultimo.

Esempio:
Supponiamo di avere un sistema con gestione di I/O con interrupts e supponiamo che ogni millisecondo arriva un byte; questo significa che ogni millisecondo la CPU sarà interrotta da un interrupt che richiederà l'esecuzione di una determinata interrupt service routine (ISR), quest'ultima potrebbe consistere, ad esempio, nella semplice lettura/scrittura del dato.

Se si dovessero trasmettere 1000 dati da 1byte ciascuno, la CPU subirebbe 1000 interrupts.

Supponiamo che ogni interrupt ha un overhead che porta al consumo di 2µs per le azoni di sospendere/caricare il foreground program e di 98µs per eseguire l'ISR, per un totale di 100µs. Per 1000 bytes (e quindi 1000 dati, ovvero 1000 interrupts), la CPU subirà un ritardo di 0.1s

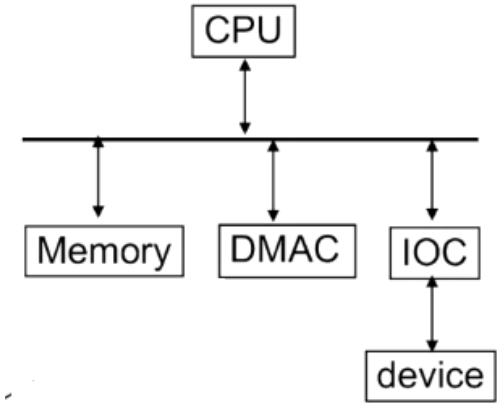
$$T_{OVERHEAD} = T_{FOREGROUND} + T_{ISR} \rightarrow T_{1000bytes} = 1000\, T_{OVERHEAD} = 1000 * [(2 + 98) * 10^{-6}] = 0.1s$$

Con un flusso dati con rate pari a 10MB/s il sistema deve "smaltire" i dati in 0.1µs per byte per evitare la perdita di informazione. Ma questo non è possibile dato che il sistema è in grado di smaltire i dati in tempi pari a 100µs. Con questo flusso di dati dovremmo fare in modo di trasferire 1000 bytes in 100µs, mentre il sistema in questo tempo riesce a smaltire solo 1 byte.

Sia nella tecnica di gestione con polling che in quella con interrupt il trasporto del dato spetta alla CPU con l'unica differenza che per l'interrupt non effettua i continui check sulla disponibilità della risorsa. Quindi deve ancora essere migliorato il trasporto del dato, che fa perdere tempo utile di CPU, trovando un grado di parallelismo maggiore, lasciando la gestione del trasporto all'I/O device: parleremo allora di tecnica DMA: Direct Memory Access.

Tecniche di gestione dell'I/O: Direct Memory Access (DMA)

Un'altra tecnica di gestione dell'I/O è il **DMA: Direct Memory Access**. Utilizzando tale tecnica si punta a minimizzare i tempi di CPU, esonerando quest'ultima dalla funzione di trasporto dei dati (a differenza delle due tecniche precedenti: Polling e Throughput).
Il DMA è una tecnica che consiste nell'utilizzo di quello che viene chiamato DMAC: DMA Controller; questo è collegato sul bus in cui vi sono collegate CPU, memoria e l'IOC: I/O Controller per il controllo dei Device di I/O.
Il DMA è programmabile: la CPU comunica, ad esempio, che quando vi sono dei dati provenienti da un determinato dispositivo di I/O, esso dovrà leggerli dal dispositivo e dovrà scaricarli in memoria a partire da quell'indirizzo per N volte.

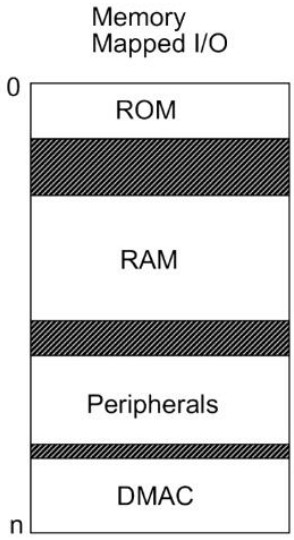


Così facendo la CPU ha perso del tempo per programmare il DMA ma da questo momento in poi, tutte le volte che arrivano dei dati dal device in questione vi sarà un trasferimento automatico in memoria, senza passare dalla CPU. Quindi da un punto di vista ideale il DMA effettua il trasferimento dal device di I/O alla memoria senza passare dalla CPU**: in questo modo (idealmente) si solleva la CPU dal trasferimento dei dati,** ovviando quindi ai tempi di CPU sprecati nel caso di Polling ed Interrupt per il trasporto dei dati**.**

Il DMA insieme al sistema di I/O ha la capacità di interfacciarsi autonomamente con la memoria, trasferendo N byte senza dover passare dalla CPU.

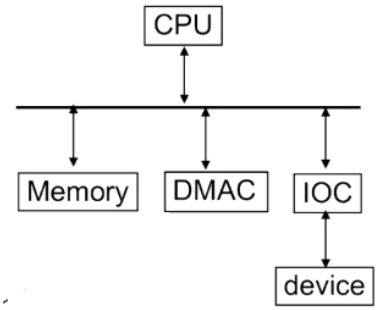
Analizziamo adesso il tempo sprecato nella gestione di I/O con DMA:

1. Il DMA instaura una set-up sequence per settare il DMAC per la gestione del trasporto dei dati dall' I/O Device alla memoria. Supponiamo che tale tempo sia:
$$T_{SETUP} = 50\mu s$$
2. Quando il DMA termina il trasferimento genera un Interrupt alla CPU che prende il nome di **interrupt end of transfer**: si hanno quindi 2µs di overhead della CPU, impiegati per stoppare il foreground program e per riprenderlo successivamente e, inoltre, 48µs impiegati dalla interrupt service routine (ISR) dell'interrupt per lanciare il DMA alla CPU.
$$T_{ISR} = 50\mu s$$
3. Quindi per il trasferimento di un byte si impiegano esattamente 0.0001 secondi dei tempo di CPU. Questo allora ci porta ad affermare che, con la tecnica di gestione DMA, possiamo gestire un traffico dati nell'ordine dei MB/s.



Si osservi però che in questo modo, se CPU e DMAC sono collegati sullo stesso bus, teoricamente, dovrebbero lavorare in modo alternato poiché potrebbero utilizzare il bus una volta a testa; quindi, non è sempre detto che i tempi impiegati siano quelli appena discusso poiché **può capitare che, in alcuni casi, DMAC e CPU entrino in conflitto per accedere al bus.**

Il DMA, quando è in funzione, si “sostituisce” alla CPU, eseguendo le operazioni di indirizzamento della CPU grazie ad un hardware che lo permette (con dei registri contatori); esso esegue le stesse funzioni che eseguirebbe la CPU nei casi precedentemente studiati: controlla se una periferica ha terminato e si occupa dell’indirizzamento (non potrà, chiaramente, effettuare alcuna computazione). Ma quindi la CPU resta ferma quando il DMA sta utilizzando il bus, questo significa che, teoricamente, non vi alcuna forma di parallelismo! In realtà non è sempre così: il conflitto nasce, ad esempio, quando la CPU deve accedere in memoria poiché per farlo dovrà passare dal bus.



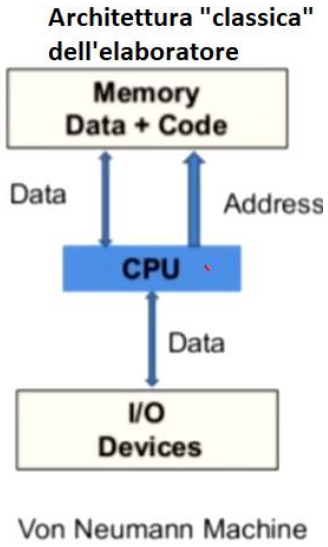
Chiaramente, la CPU dovrà sempre accedere in memoria per eseguire le istruzioni, quindi parleremo di un’architettura del calcolatore che risolve questo problema è l’architettura **Harvard.**

Architettura Harvard

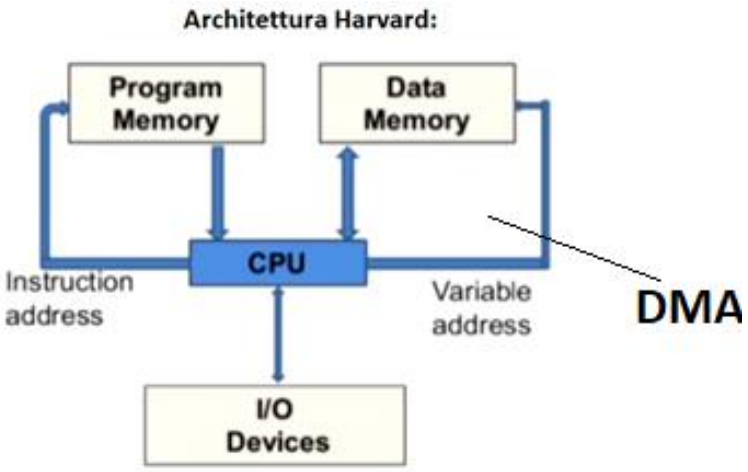
Nella classica architettura a cui siamo abituati, quella di **Von Neumann** viene utilizzata un’unica memoria che viene sfruttata sia per il traffico dei dati che per l’esecuzione delle istruzioni.

Un’architettura differente è l’architettura Harvard che prevede l’utilizzo di due memorie: una per le istruzioni, la **Program Memory** e l’altra per i dati, la **Data Memory,** quindi la CPU accede ad esse con due bus differenti.

Con architettura Harvard si risolve il problema della mancata parallelizzazione che abbiamo riscontrato utilizzando il DMA nell’architettura di Von Neumann.
Nell’architettura Harvard, infatti, il DMA sfrutta il bus per accedere alla Data Memory, mentre la CPU sfrutta un altro bus: quello per accedere alla Program Memory.



Von Neumann Machine



Harvard Machine

Tuttavia, è ancora possibile avere un conflitto: la CPU può andare in parallelo al DMA solo se le istruzioni che devono essere processate non sono di **load/store**;
sommariamente, queste istruzioni non sono frequenti: si parla di circa il 15-20% delle istruzioni totali, quindi per circa l’80-85% di istruzioni l’architettura Harvard offre il parallelismo richiesto.

Domande su Interfaccia Cyber-Physical:

1. In un sistema di interfaccia Cyber-Physical in cui sono presenti N sorgenti analogiche, ciascuna caratterizzata da $f_{i\,MAX} = K_i$ [frequenza massima del segnale dell'i-esima sorgente], un MPX analogico N:1 e a valle del MPX un Sample & Hold e un unico convertitore A/D. Dire quale delle seguenti opzioni è vera:
   a. Il massimo fra i tempi di campionamento T_{Ci}, con i = 1,...,N determina il numero di tempo di conversione del convertitore A/D
      Falso: Il minimo dei tempi di campionamento è quello che determina il numero di tempo di conversione: più basso è il tempo di campionamento e maggiore è la frequenza di campionamento e quindi minore deve essere il tempo di conversione del convertitore.

   b. Il tempo di conversione del convertitore A/D deve essere almeno minore del minimo tra i valori $\{1/f_{i\,MAX}\}$ [= $T_{C\,i\,MAX}$ massimo tempo di campionamento], con i = 1,…,N
      Falso: Per il teorema di Nyquist-Shannon, la frequenza di campionamento deve essere maggiore o uguale al doppio della f_{MAX} ; di conseguenza, dato che il tempo di campionamento è pari all'inverso della frequenza di campionamento, avremo che $T_c \leq 1/2f_{MAX}$; quindi non deve essere "almeno minore del minimo tra i valori $1/fi_{\,MAX}$", bensì deve essere almeno minore del minimo tra i valori $1/2f_{i\,MAX}$

   c. **Il tempo di conversione del convertitore A/D $T_c < 1/2K_{MAX}$ dove K_{MAX} = max $\{f_{i\,MAX}\}$ con i = 1,…,N**
      Vero: Dal teorema di Nyquist-Shannon, la frequenza di campionamento deve essere tale che $f_c \geq 2f_{MAX}$ che nel nostro caso diventa $f_c \geq 2K_{MAX}$; dato che il tempo di campionamento è l'inverso della frequenza: $T_c = 1/f_c$ possiamo affermare che deve valere: $T_c \leq 1/2f_{MAX}$ ovvero: $T_c \leq 1/2K_{MAX}$.

   d. Nessuna delle precedenti

2. In un sistema IoT in cui sono presenti N sorgenti analogiche tutte caratterizzare dalla frequenza massima $f_{i\,MAX} = K$, con i = 1,…,N , il tempo di conversione T_C del convertitore A/D a valle del MPX analogico N:1 deve:
   a. Essere minore di 1/2K
      Falso: Per il teorema di Nyquist-Shannon, la frequenza di campionamento f_C deve essere maggiore o uguale a 2K, di conseguenze T_c , inverso di f_C deve essere minore di 1/2K. Tuttavia questo vale per una sola sorgente.

   b. **Essere minore di 1/2NK**
      Vero: rispettando il teorema di Shannon-Nyquist, per N sorgenti la f_C deve essere maggiore o uguale a $N*(2f_{MAX})$ [= 2NK], di conseguenza T_C deve essere minore di 1/2NK

   c. Essere minore di 1/NK
      Falso: il teorema di Nyquist-Shannon impone che la f_C sia maggiore o uguale al doppio della f_{MAX} ; quindi per essere vera si necessita del doppio di NK al denominatore.

   d. Nessuna delle precedenti