# Lezione 38 Input e output 2

venerdì 8 marzo 2024 13:03

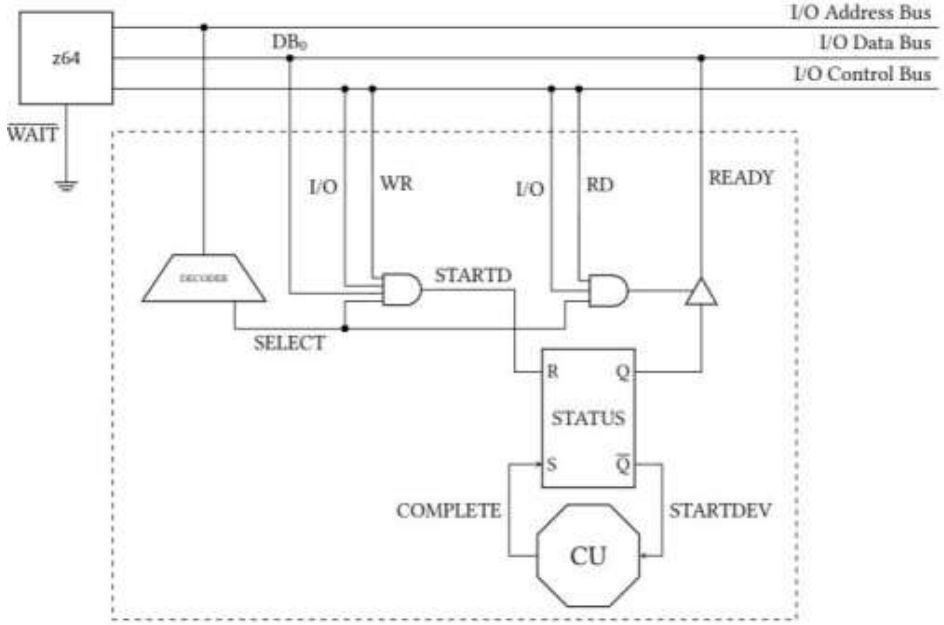
Andiamo ora a vedere i tipi di interazione tra cpu e dispositivi : cominciamo con **I/O programmato** : nel quale si ha che è il programma sw che inizializza e governa le interazioni con i dispositivi per effettuare il trasferimento dei dati , poi vediamo **richiesta esterna** : il dispositivo richiede attenzione della cpu , la quale esegue opportuno driver per gestire la richiesta ed il trasferimento dei dati, infine invece si ha : **interazioni gestite da processori dedicati** : il processore trasferisce la responsabilità della richiesta e/o gestione ad un coprocessore. Dedichiamoci in dettaglio all'i/o programmato : in questa interazione è il software che guida lo scambio di dati tra periferiche e cpu , quindi il software necessita di particolari istruzioni per poter effettuarla , le quali sono :

| Instruction | Syntax | Semantics |
|---|---|---|
| Inbound transfer from parametric I/O port | `inX %dx, RAX` | Transfer data of size X from the device deployed on the I/O address contained in the %dx register. |
| Inbound transfer from explicit I/O port | `inX $ioport, RAX` | Transfer data of size X from the device deployed on the I/O address `$ioport`. |
| Outbound transfer to parametric I/O | `outX RAX, %dx` | Transfer data of size X to the device deployed on the I/O address contained in the %dx register. |
| Outbound transfer to explicit I/O | `outX RAX, $ioport` | Transfer data of size X to the device deployed on the I/O address `$ioport`. |

Dove %dx contiene fino ad un massimo di 2^16 porte : ovvero questo registro contiene al massimo 2 ^16 possibili indirizzi. Questo tipo di interazione avviene secondo due tecniche : **busy waiting e polling** . Per quanto riguarda il primo si ha che avviene attraverso 4 passi :

1. Il processor avvisa il dispositivo che vuole effettuare un trasferimento , secondo il modello vista prima.
2. Il preprocessore verifica il bit di status ( bit del ff) se è resettato(0) oppure settato(1)
3. Se il ff (status) è resettato , il preprocessore continua ad aspettare , mettendosi nello stato di wait
4. Il programma continua il suo normale flusso

Quindi in questa modalità non è possibile esplicitamente interazione tra bus e dispositivi , almeno che non è la cpu ad interrogarli direttamente, quindi non vi è più del famoso segnale di wait negato : viene quindi collegato a massa . Sembra efficiente questa modalità ma in realtà soffre delle stesse problematiche dell' i/o implementato a firmware , ed in particolare il processore non esegue nessun'altra attività fino al completamento del trasferimento dei dati.
Quindi andiamo a rivisitare l'architettura per i/o :




Da notare in questa architettura : innanzitutto il segnale di wait negato viene messo a massa, levando cosi la tecnologia open-collector ( porte and per ottenere or logico) , poi in base and indirizzo sul bus e decoder , viene selezionato un solo device , il quale device prende il dati dal

data bus, ed abilitando i segnali di i/o e wr (scrittura su device) , va a resettare il ff (status) che essendo 0 , il processore aspetta finche' non viene settato(1).Per quanto invece riguarda la lettura il procedimento è uguale : si parte da indirizzo-> decoder->1 solo dispositivo, poi abilitando il segnale i/o e lettura , il dispositivo va a leggere i dati dal bus ed attiva o meno un buffer tri state per dire al processore che ha finito. Per quanto riguarda i segnali invece :
**STARTD:** Questo segnale identifica la volontà del processore di avviare un protocollo di scambio dati con il device. Nello specifico questo segnale resetta il FLIP FLOP di status.
**STARTDEV**: È il buffering del segnale STARTD. Lo scopo di questo segnale (che diventa un ingresso della CU del dispositivo) è quello di generare una transizione di stato affinché l'operazione di interesse (produzione o consumazione dati) possa essere avviata. **COMPLETE**: Questo è il segnale di controllo generato dalla CU del dispositivo. Questo segnale imposta il valore del FLIP FLOP STATUS ad 1, indicando che l'operazione richiesta dalla CPU è stata eseguita. In dettaglio (almeno in assembler e concettualmente) :

```
        movb $1, %al
        outb %al, $device
.bw:
        inb $device, %al
        btb $0, %al
        jnc .bw
```

- Problema: il processore esegue lo stesso codice finché non è completato il trasferimento
- Nonostante il processore non sia in *stallo*, non vengono effettuate attività utili
- Possibile soluzione: interrogare altre periferiche e servire la prima disponibile

Andiamo ora a vedere la seconda tipologia : **il polling :** è simile al busy waiting , in quanto effettua una verifica circolare per vedere la prima periferica pronta ad interagire . Vediamolo in dettaglio :

```
.poll :
    movw $STATUS_DEV1, %dx
    inb $STATUS_DEV1, %al
    btb $0, %al
    jc .dev1
    inb $STATUS_DEV2, %al
    btb $0, %al
    jc .dev2
    # ...
    jmp .poll

. devX:
    # ...
    jmp .poll
```

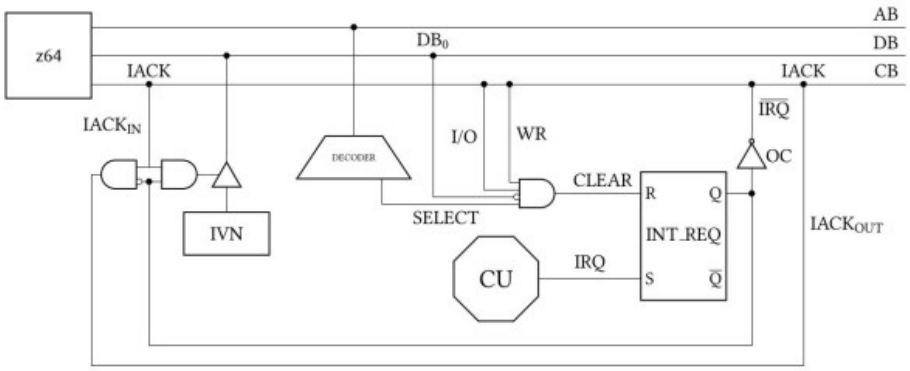
- Problemi: il processore deve continuamente testare il valore di STATUS
- Codice difficile da manutenere: cosa succede se si aggiunge un dispositivo?
- Rischio di *starvation*: le ultime periferiche possono non essere servite mai
- Difficile gestire le *priorità*

Andiamo ora a vedere come alterare il flusso del programma , magari per far si' che il flusso ne sia alterato : le **interruzioni**: meccanismo che forza il processore ad interrompere il normale flusso di esecuzione in favore di un altro flusso di esecuzione : quindi per fare si che ciò avvenga, vi deve essere eterogeneità tra i dispositivi interconnessi alla cpu. Quindi per fare si che ciò avvenga, i dispositivi alzano una "sorta di bandierina bianca" , e quando questa viene riabbassata( richiesta eseguita) ,il processore riprende la normale esecuzione. Questa gestione è priva di rischi?? Assolutamente no , in quanto la generazione di un'interruzione(alzo bandierina bianca) è *asincrona* , più dispositivi allo stesso istante richiedono attenzione cpu, il processore deve vedere quale dispositivo ha sollevato interruzione per poterlo gestire correttamente( driver) , ed infine la cpu può eseguire un solo driver alla volta . Quindi come possiamo gestirle in modo corretto? O ad ogni interruzione gli diamo una priorità (**daisy chain)** oppure realizzare un protocollo per identificare i dispositivi che hanno sollevato l'eccezione. Prima di addentrarci in queste gestioni , vediamo come si gestisce una richiesta di interruzione:

1. Si deve salvare lo stato attuale del programma (registri)
2. Identificare il driver per gestire in modo corretto il dispositivo che ha generato interruzione
3. Esecuzione del driver

4. Ripresa della normale esecuzione del programma

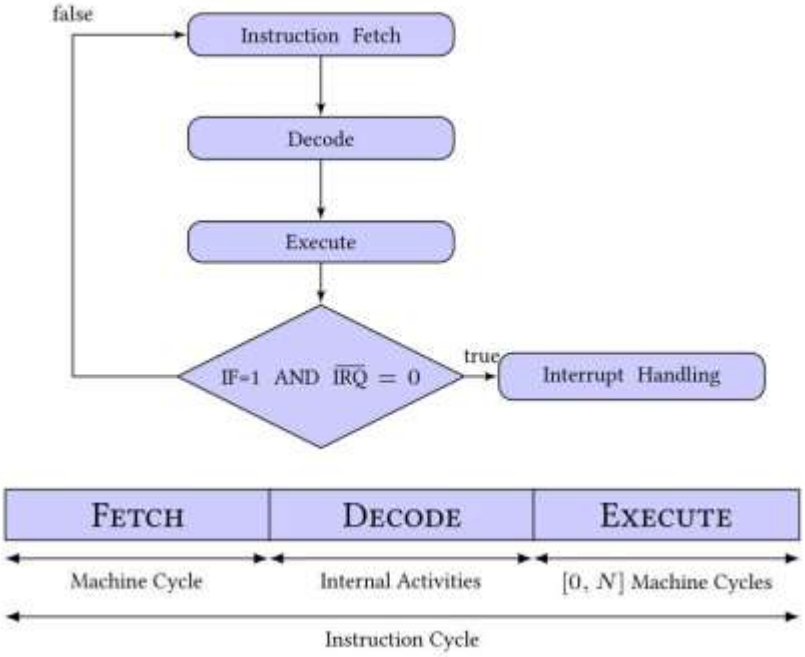
Andiamo a vedere il **daisy-chain**:



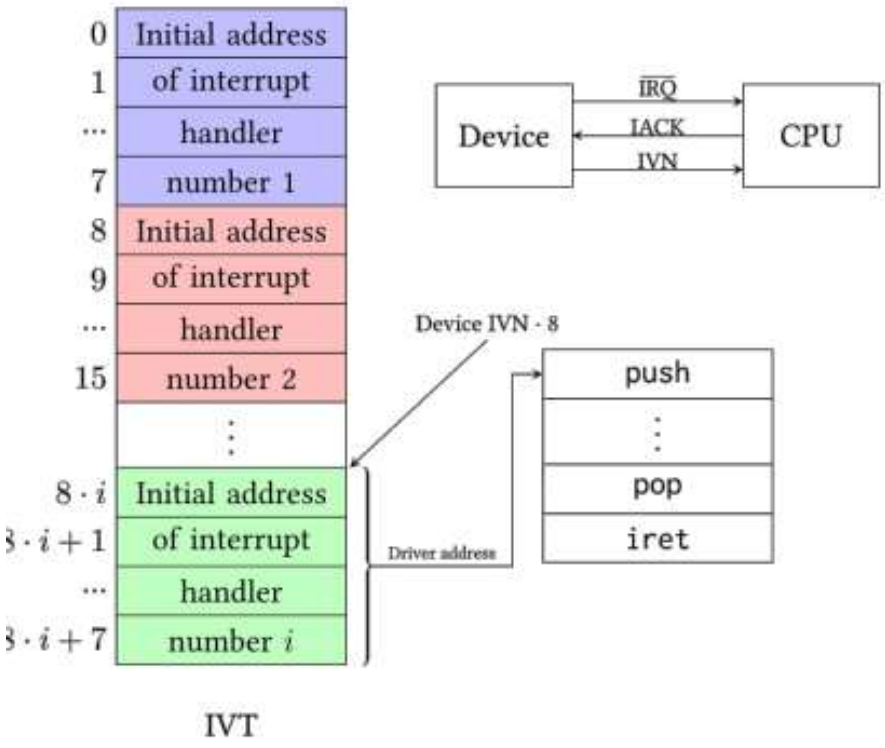

In questo schema il processore utilizza un segnale particolare (iack)->interrupt ack per dire che è pronto a servire una richiesta d interruzione . Analogamente a prima , per scegliere il dispositivo da servire ( con relativo driver) si va ad usare l'indirizzo preso e codificato dal decoder per scegliere il dispositivo ( più vicino è ad indirizzo e più sarà servito rapidamente) , poi abilitando i segnali di i/o e WR si va a fare l'interazione. Per quanto riguarda il FF(stato interno) viene chiamato interrupt-request il quale viene posto a 0 (reset) se il dispositivo/processore ha eseguito il driver e questo segnale torna sul bus controllo attraverso logica di open-connector( and che si comportano da or) . Tornando invece al segnale di iack , prima di tornare sul bus di controllo , va in and con l'uscita del ff (interrupt-request) , e torna sul bus , ma se interrupt-request settato (1) il processore scrive altrimenti no( notare che q in entrata alla and è negato). Cosi funziona la gestione dell'interruzione, ma come fa il processore a scegliere quale driver usare per gestire?? In base all'**IVN ( interrupt vector number):** in base al numero si attiverà un'opportuna funzione di gestione(driver). Vediamo un semplice esempio (eseguito su un processore multiciclo):

```
movq $0, %rax
testq %rax, %rax
```

Una richiesta di interruzione potrebbe essere ricevuta o durante il movq oppure tra movq e testq (test se registro è 0) , quindi in generale potrebbe succedere che : i registri invisibili al programmatore potrebbero essere corrotti , registro flags corrotto oppure il contenuto del registro %rax è stato corrotto. Quando da normale esecuzione del programma , si passa alla gestione del driver (basato su IVN) , si ha a che fare con quel meccanismo che prende il nome di cambio di contesto : prima salva lo stato del programma (registro RIP ->indirizzo da cui riprendere esecuzione, registro FLAGS bit di condizione , registri invisibili al programmatore Temp1/Temp2 , registri general purpose ) su stack -> interrupt frame , poi attiva il programma per la gestione del driver. Attenzione però al costo : se architettura troppo complessa , il costo sale!! In generale nel processore ci deve essere un meccanismo , che impedisce al processore di seguire più gestioni di interruzioni contemporaneamente /anche la stessa (gestione attività atomica). Capita spesso che il processore deve "mascherare" l'ivn a 0 : quindi il processore va a modificare il *if (interrupt flag)* , attraverso lo sti/cli . Questo flag viene azzerato dal firmware durante il cambio di contesto. In virtù di quanto detto finora, vediamo ora come preservare i registri **soprattutto RIP e FLAGS**:




**Il microprogramma viene eseguito <u>completamente</u> cosi' non si ha la corruzione dei registri invisibili al programmatore (registri salvati in modo consono).** La gestione fatta finora delle interruzione sembrerebbe bastare , ma in realtà no in quanto il processore non sa quale è realmente il driver da eseguire : manca il collegamento tra IVN (numero che identifica il dispositivo) e lo usa come indice di vettore di puntatori , e si spiazza di 8*ivn e IVT( interrupt vector table ): tabella che contiene riferimenti a prime istruzioni per la gestione opportuna del driver:




Vediamo ora come avviene a livello istruzioni microcodice:

| | |
|---|---|
| FLAGS[I] ← 0; TEMP1 ← RSP<br>TEMP2 ← 8<br>RSP ← ALU_OUT[SUB]<br>MAR ← RSP<br>MDR ← RIP<br>(MAR) ← MDR<br>TEMP1 ← RSP<br>TEMP2 ← 8<br>RSP ← ALU_OUT[SUB]<br>MDR ← FLAGS<br>MAR ← RSP<br>(MAR) ← MDR<br>IACK<br>IACK; MDR ← IVN<br>TEMP2 ← MDR<br>MAR ← SHIFTER_OUT[SX, 3]<br>MDR ← (MAR)<br>RIP ← MDR | Setto if a 0-> niente interruzioni; mi salvo RSP in temp1<br>Decremento stack<br>Rsp lo salvo nello stack<br><br>Push del contenuto di flags<br><br>Segnale di interruzione<br>Ivn viene scritto sul bus<br><br>Shifto a sx mdr di 3 posizioni (indirizzo da cui leggere indirizzo prossima istruzione)<br>Prendo il valore dell'indirizzo iniziale del gestore driver<br>Fetch prima istruzione |
| MAR ← RSP<br>MDR ← (MAR)<br>FLAGS ← MDR<br>TEMP1 ← RSP<br>TEMP2 ← 8<br>RSP ← ALU OUT[ADD]<br>MAR ← RSP<br>MDR ← (MAR)<br>RIP ← MDR<br>TEMP1 ← RSP<br>TEMP2 ← 8<br>RSP ← ALU OUT[ADD]<br>FLAGS[I] ← 1 | cima stack<br>Leggo cima stack<br>Sposto stack in flags<br><br>Incremento stack di cella<br>Mi sposto nella cella<br><br>Sposto il contenuto in mar<br>Nel bus ci metto il contenuto<br>In rip ci metto indirizzo prossima istruzione<br><br>Incremento stack<br>Metto ad 1 if (interrupt flags) |

Ricordiamo che x0800 (.org 0x800) perché questa è la taglia dell'interrupt vector. Ricordiamo anche che : **l'esecuzione di un qualunque driver avviene nella sua interezza prima di restituire il controllo (atomicità)**. I driver della periferica si dividono principalmente in due :

1. Top half : parte di lavoro non rinviabile eseguita con if=0
    a. Recupero dei dati dal dispositivo
    b. Se necessario riprogrammazione della periferica
2. Bottom half : parte rimandabile a priorità inferiore
    a. Processamento dei dati
    b. Può essere eseguita anche con le interruzioni abilitate
    c. Sti (istruzione esplicita)

Vediamo un esempio :

```
. driver 1
        # save registers clobbered in the driver
        pushq %rax
        # read the data coming from the device
        # make a temporary copy in memory
        inw $device_reg, %ax
        pushw %ax
        # delete the cause of the interrupt
        movb $0, %al
        outb %al, $device_irq
        # enable reception of interrupts : start of the bottom half
        sti
        # do any action to process the acquired data
        popw %ax
```

```
# restart the device to produce new data
# (if the device is supposed to be used like this)
movb $1, %al
outb %al, $device_status
# Destroy the interrupt frame and return control to the
# interrupted program
popq %rax
iret
```

<u>Attenzione : è possibile che un dispositivo operi sia in busy waiting che in asincrona , ma alla fine è il processore che sceglie quale modalità : arrivando cosi' alla modalità mista</u> :