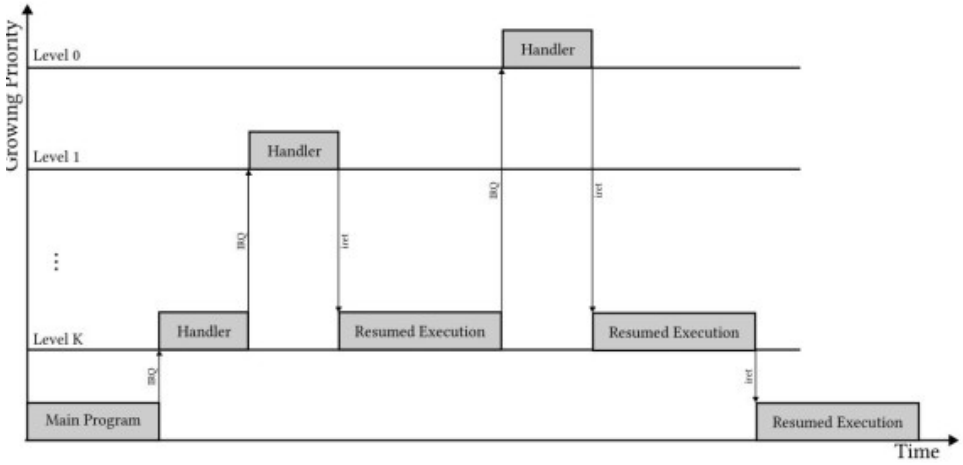
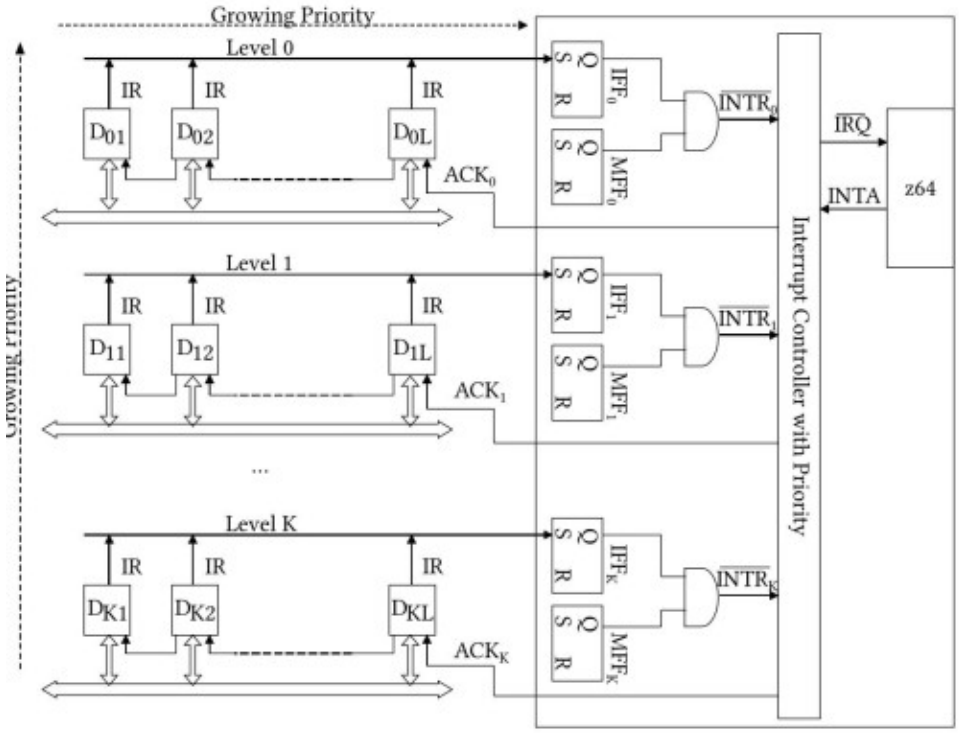
Lezione 39 Input e output 3

venerdì 8 marzo 2024 17:31

Focalizziamoci ora sulla priorità , andando a vedere i due più comuni problemi che possono sorgere : **Priority inversion e starvation.** La prima consiste nel cambiare la priorità di un dispositivo : se un dispositivo effettua un'interruzione (quindi il processore attiva il driver) , almeno che nel driver vi sia istruzione esplicita si sti (cambia interruzione) , questo non può essere interrotto . Mentre la seconda si ha quando magari c'è un dispositivo a priorità alta e molto veloce , e siccome il processore serve solamente ed esclusivamente questo dispositivo, magari un dispositivo a priorità minore non verrà mai servito (neanche con schema daisy chain). Supponiamo di avere livelli di priorità : almeno con questa soluzione si può interrompere esecuzione driver priorità maggiore , eseguendo driver di quello priorità minore.

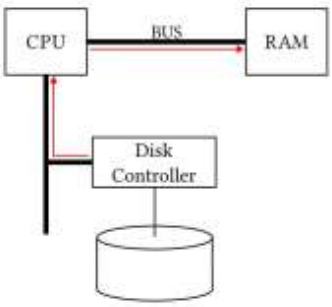


Sembrerebbe funzionare, ma attenzione al ciclo infinito causato dal bit di flag . Quindi si cerca un trade-off : organizzo i nostri dispositivi in determinati classi di priorità :

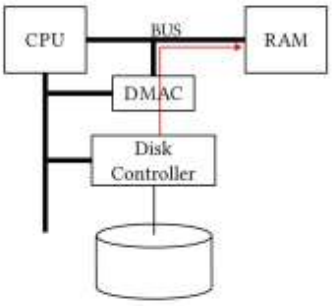


Per ogni dispositivi organizzati in daisy chain e per ogni classe di priorità vi è un solo segnale di interrupt request , in modo che il processore attivi il driver associato ad uno di questi dispositivi (stessa classe). In generale per qualunque richiesta di interruzione (qualunque classe), il processore non deve più mascherare il bit, ma può benissimo eseguire altre richieste **di classi superiori**, ignorando cosi le richieste multiple di uno stesso livello. **Quindi con questa soluzione si risolve l'inversione di priorità**. Per quanto riguarda invece la starvation un modo

possibile è il seguente : è possibile dare più priorità a classe minore , usando/settando in modo opportuno i ff (memory mapped) , mascherando cosi le richieste delle classi a livello inferiore. Quindi vista l'efficienza (non buona ma insufficiente) , si opta per usare un canale dedicato , che nonostante i due cicli per trasferire data (1 per il fetch ed altro per trasferimento) , si ha che comunque non basta : in caso di trasferimento di mole di dati molto grandi è lenta :

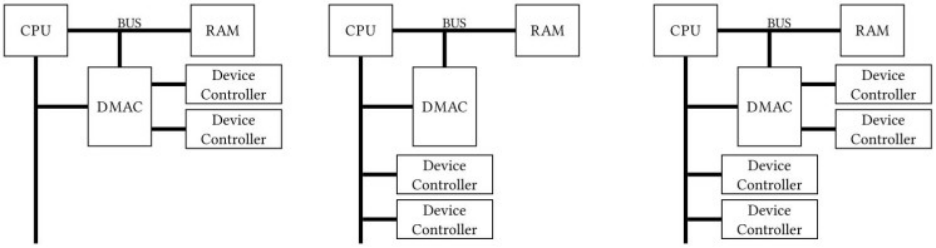


Accesso a memoria indiretto



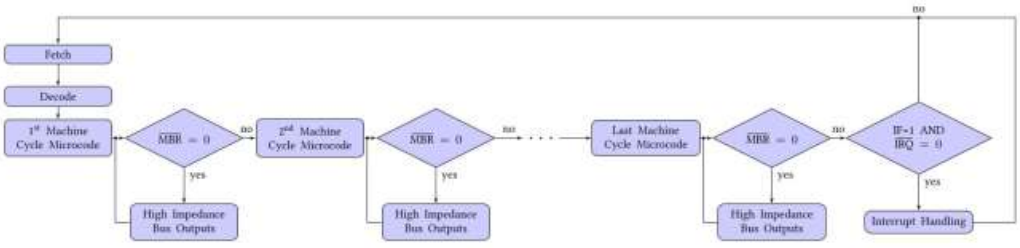
Accesso a memoria diretto

Quindi si opta per una soluzione più efficiente: inserisco un coprocessore (controllore di bus) che effettua trasferimenti tra dispositivi e memoria a posto della cpu : si parla di **DMAC(Direct memory access control)** :

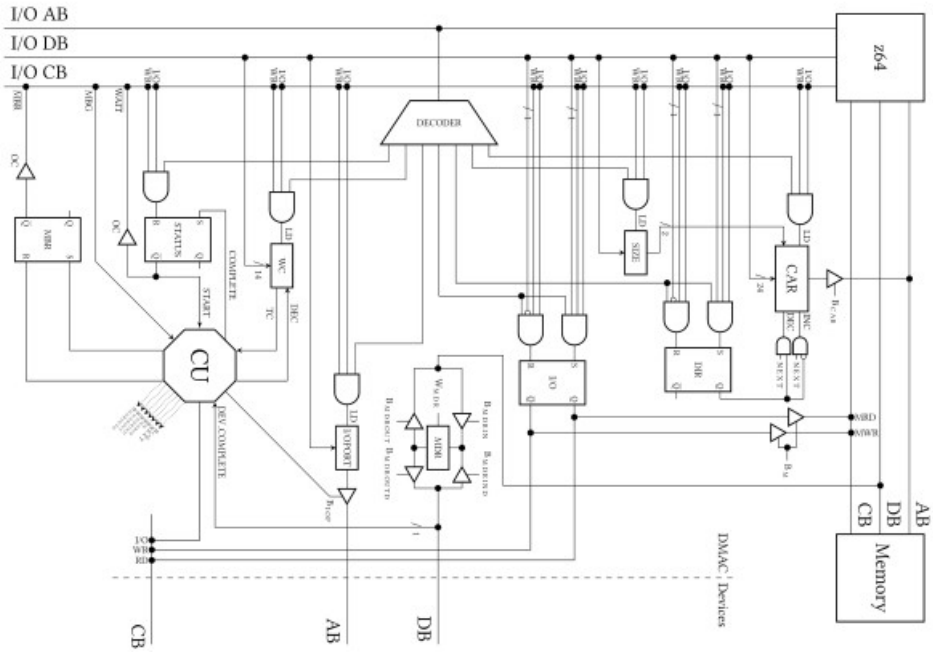


Periferiche invisibili alla CPU Periferiche condivise con la CPU Modalità mista

Nel primo modo si ha un vantaggio quando ci sono dispositivi lenti , la seconda si ha quando le periferiche sono condivise tra processore e dmac , mentre nell'ultima quelli più lenti solo dal dmac , mentre per quelli più veloci o processore o dmac : questa scelta viene fatta dal sw .**Anche il dmac viene visto come dispositivo programmabile. Ma attenzione al bus condiviso : se entrambi inviano dati su stesso bus esplode tutto .** Quindi per evitare di fare ciò si deve progettare un meccanismo che permette di coordinare CUP e DMAC . Ma come si fa ? Costruiamo un protocollo di handshaking tra loro : mutua esclusione dell'utilizzo del bus condiviso : usiamo due segnali di controllo : **MBR (memory bus request)** : utilizzato dal dmac per effettuare un trasferimento . Lavora in logica negata (tecnologia open collector) : per evitare più richieste in parallelo (quindi più dmac lavorano in parallelo) ed il processore usa il **MBG (memory bus grant):** segnale di alta impedenza che permette di staccarsi dal bus di sistema (isolamento) : in sintesi con questo segnale si dice che il dmac ha controllo assoluto sul bus . Finchè MBR negato vale 1 (asserito) il processore è in stallo. Quindi in generale si deve minimizzare il tempo che il dmac invia dati (processore isolato). Vediamo ora come minimizzare :due tecniche **burst mode e bus stealing**. Nella prima quando il processore asserisce MBG , il dmac mantiene MBR negato asserito , finchè non ha completato il trasferimento dei dati : modalità inefficiente perché se grande quantità di dati si ha sottoutilizzo processore. Per migliorarla si passa alla seconda , la quale il dmac richiede il bus solo quando ha un dato da trasferire :dato a 32 bit immagazzinati nel registro tampone . Quindi effettuando questo trasferimento a pezzi di 32 bit, entrambi i dispositivi riescono a usare il bus condiviso .



Quindi il circuito è il seguente:



Notiamo che CAR (current address register) mantiene l'indirizzo corrente verso quale scrivere/leggere. Può essere scritto direttamente dal processore dello z64. Massimo 2^24 bit indirizzi (separazione degli indirizzi). Notiamo anche il registro size : specifica la grandezza dei byte da copiare (leggere/scrivere) : byte, word, long word, quad word a partire dal car. Quindi CAR è una sorta di registro contatore ! Il quale usa anche il df (direction flag) copiato all'interno del ff dell'interfaccia (DIR) (per sapere dove scrivere/leggere) . WC (word count) è un registro che dice quante parole di una determinata taglia dobbiamo trasferire (avanti/indietro) a partire da CAR. WC se arriva a 0 fornisce TC (terminal count) : trasferimento completato : analogo allo zf (zero flag). Per quanto riguarda invece il dispositivo con cui dobbiamo interagire : prendo indirizzo e lo scrivo in porta i/o (registro) ; per scegliere invece il tipo di trasferimento lo sceglie il sw scrivendo il valore all'interno di registro di interfaccia (I/O) , il quale permette MRD e MWR (verso memoria) oppure sul control bus per interazione con dispositivi. Quindi riassumendo il tutto : il processore deve scrivere indirizzo da cui partire , la direzione in cui effettuare la scrittura, la taglia del singolo blocco , se è operazione di i/o, quante parole che devono essere scritte . Il ff di status mantiene tutte le info. Vediamo ora le istruzioni che permettono di fare ciò :

Instruction	Syntax	Semantics
Inbound transfer of a data string	`insX`	Transfer an arbitrarily large buffer of data from a device.
Outbound transfer of a data string	`outsX`	Transfer an arbitrarily large buffer of data to a device.

Sono istruzioni di tipo *stringa*

`insX`: leggi 10 byte da DEV

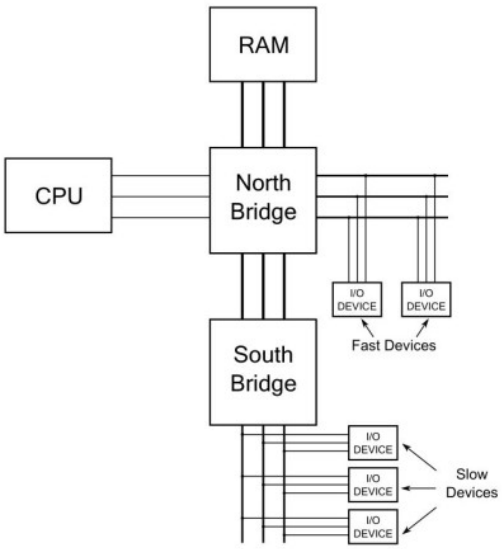
```
1 movq $10, %rcx
2 movq $dest, %rdi
3 movq $dev_mem, %dx
4 cld
5 insb
```

`outsX`: scrivi 10 byte su DEV

```
1 movq $10, %rcx
2 movq $dest, %rsi
3 movq $dev_mem, %dx
4 cld
5 outsb
```

L'esecuzione del trasferimento è *sincrona*: il DMAC asserisce $\overline{\text{WAIT}}$

Quindi : le architetture più moderne sono viste come **adattatori di bus** **(coprocessori che implementano interfacce per effettuare trasferimenti a velocità differenti)**: i dispositivi vengono organizzati per classi di velocità , ognuna delle quali gestisce una classe di dispositivi con stessa velocità . Quindi per adattatori di bus si intende dei coprocessori attaccati ai bus , i quali si dividono in due : north e south : i primi a velocità maggiori , i secondi a velocità minori . In dettaglio :


RAM
CPU
North Bridge
South Bridge
I/O DEVICE
I/O DEVICE
Fast Devices
I/O DEVICE
I/O DEVICE
I/O DEVICE
Slow Devices