Input e output

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it

Legge di Amdahl

- L'incremento delle prestazioni delle CPU è di circa il 60% ogni anno
- Le periferiche esterne sono limitate da ritardi *meccanici* o *fisici*
  - incremento delle prestazioni < 10% all'anno
- Ogni programma può essere diviso in due parti:
  - una parte f passata in attività di processamento da parte della CPU
  - una parte $1 - f$ passata in interazione con dispositivi

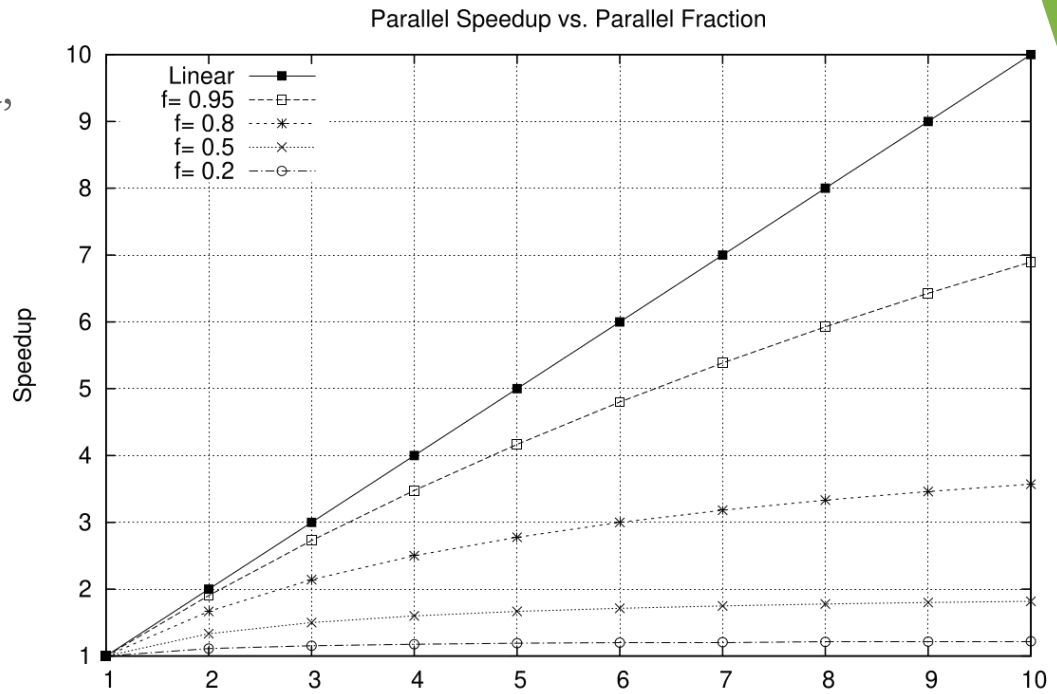
- Anche se abbiamo un processore K volte più veloce, lo *speedup* massimo è dato da:

$$S = \frac{1}{(1 - f) + \frac{f}{K}}$$

Legge di Amdahl

- I dispositivi di I/O determinano un collo di bottiglia prestazionale
- Anche se la parte f è elevata, lo speedup sarà limitato dall'I/O
- Ad esempio, se $f = 80\%$:

$$\lim_{K \to \infty} S = \frac{1}{1 - 0.80} = 5$$



Caratteristiche dei dispositivi di I/O

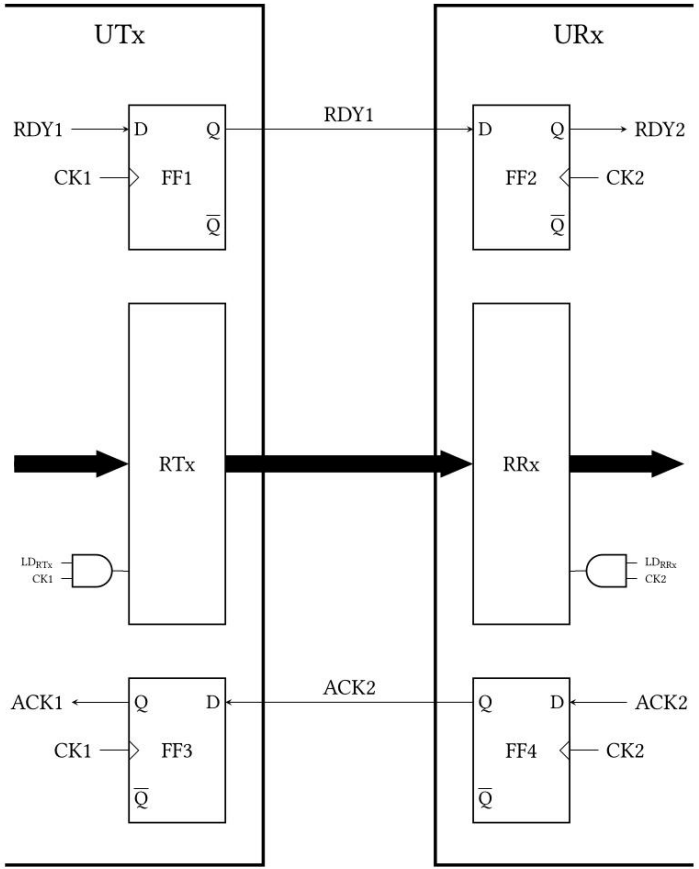
- La velocità dei dispositivi di I/O, inoltre, tende ad essere molto limitata

Class	Working data unit	Exchange speed
Magnetic Tapes	Byte	Up to 30 Mcar/sec
Magnetic Disks	Byte	Up to 300 Mcar/sec
Serial Printers	Byte	200–1200 car/sec
Parallel Printers	Byte	1k–100 kcar/sec
CRT Screens	Byte	300–19.2 Kcar/sec
Analog/Digital Converters	8–16 bits words	10–10 Mwords/sec
USB 1.0	Byte	1.5 Mcar/sec
USB 2.0	Byte	60 Mcar/sec

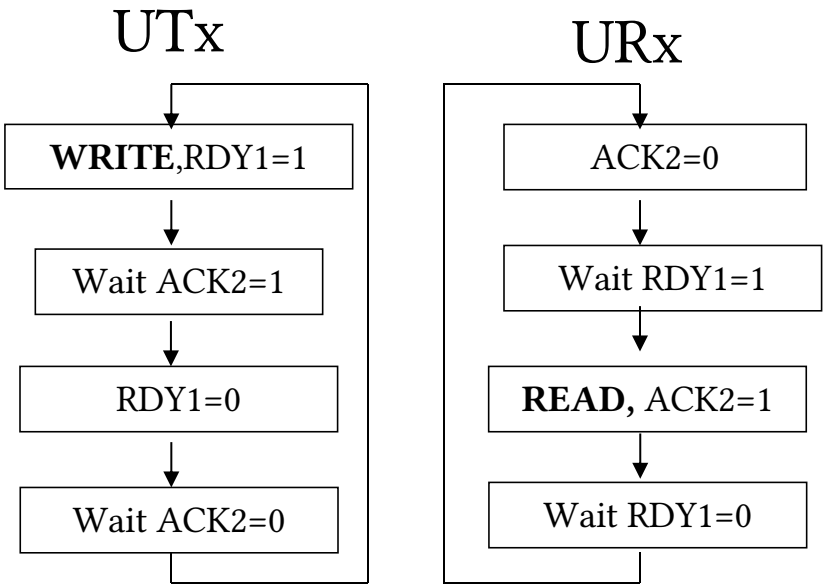
Protocollo di handshaking

- Le velocità più basse dei dispositivi richiedono una “sincronizzazione momentanea” la CPU e i dispositivi
- L’obiettivo è sempre quello di garantire la stabilità dei dati tra le reti combinatorie
- Non è pensabile che il processore rallenti per aspettare il dispositivo, né che il dispositivo operi più velocemente
- Un insieme di registri tampone e di segnali di controllo permette una sincronizzazione e un trasferimento dati corretto
  - Richiesta di svolgere operazioni
  - Produzione di dati
  - Notifica che i dati prodotti sono stati processati

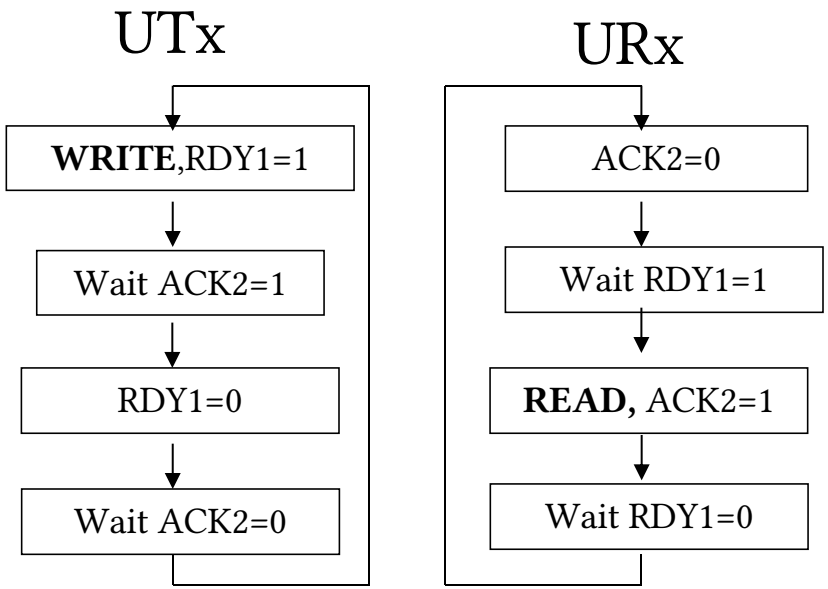
Protocollo di handshaking



Valori iniziali: RDY1=0, ACK2=0

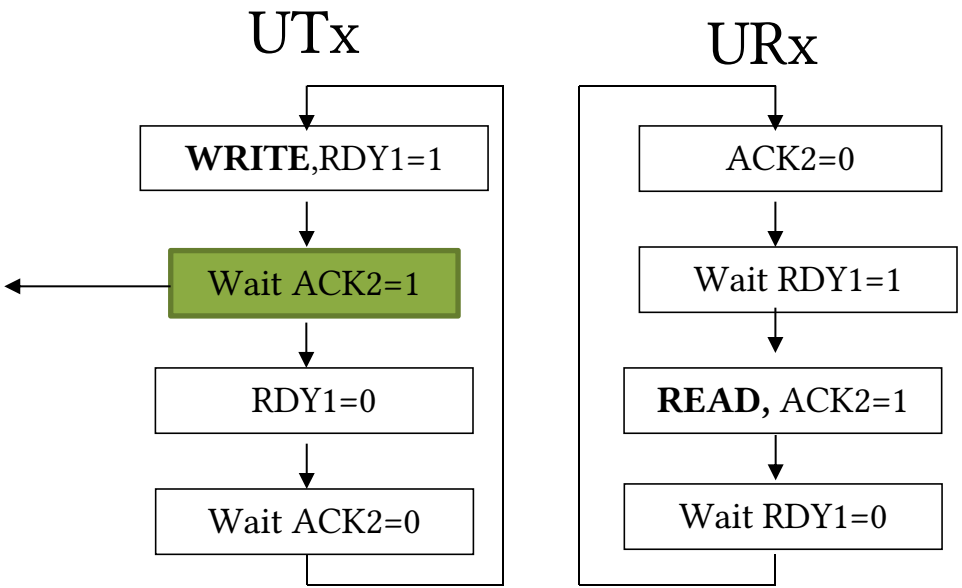


Cosa succede se si rimuovono le wait

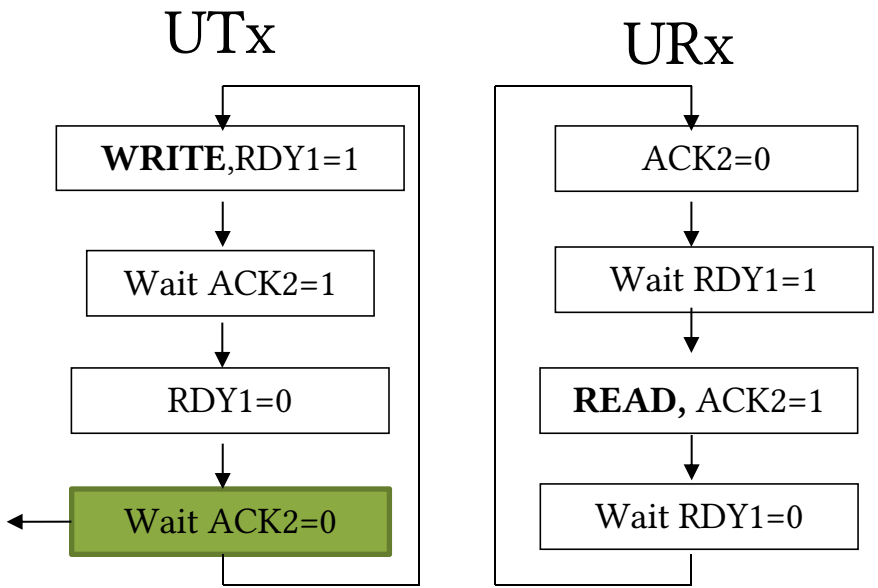


Cosa succede se si rimuovono le wait

Sovrascrivo prima che la lettura sia avvenuta

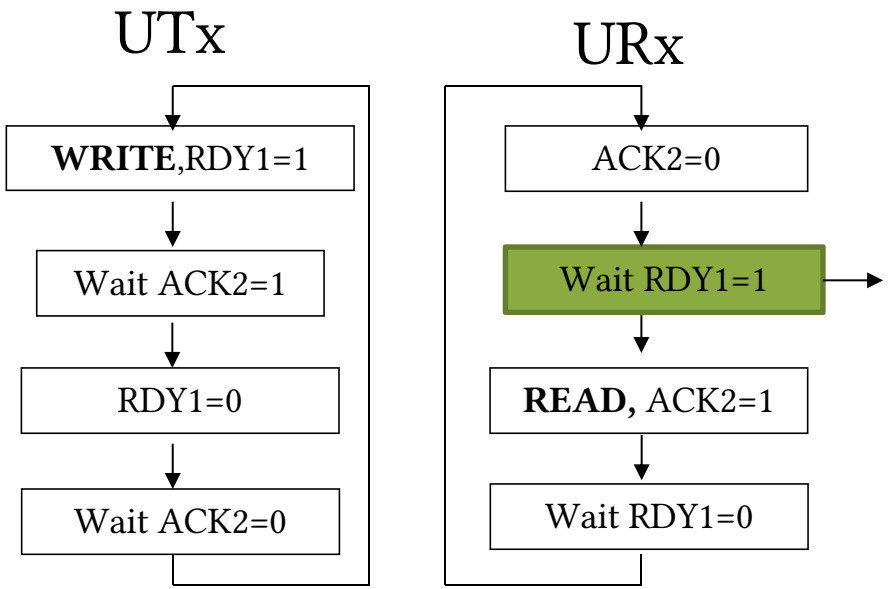


Cosa succede se si rimuovono le wait



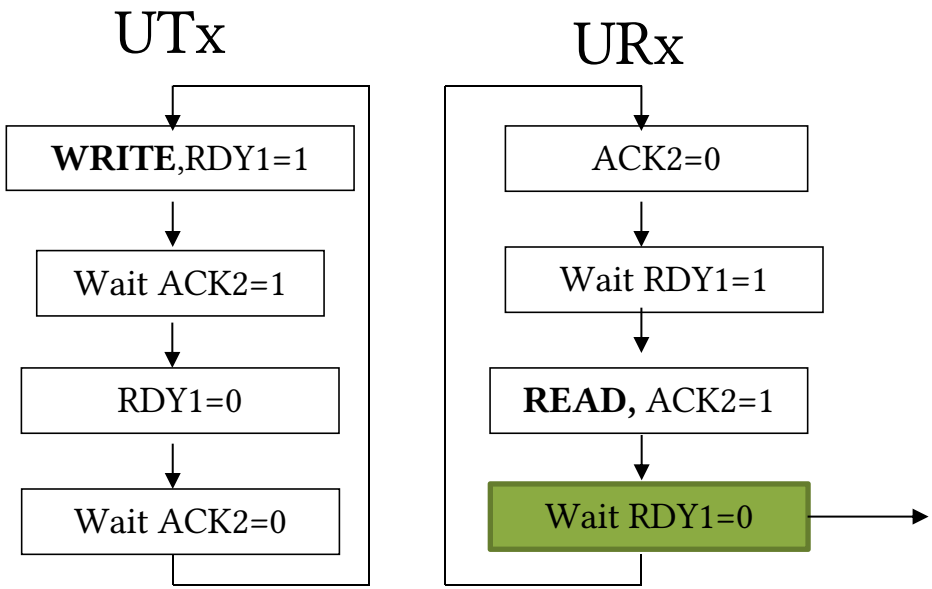
Al ciclo successivo alla scrittura potrei trovare ACK=1 dal ciclo precedente, senza che la lettura del nuovo valore sia avvenuta: sovrascrittura

Cosa succede se si rimuovono le wait



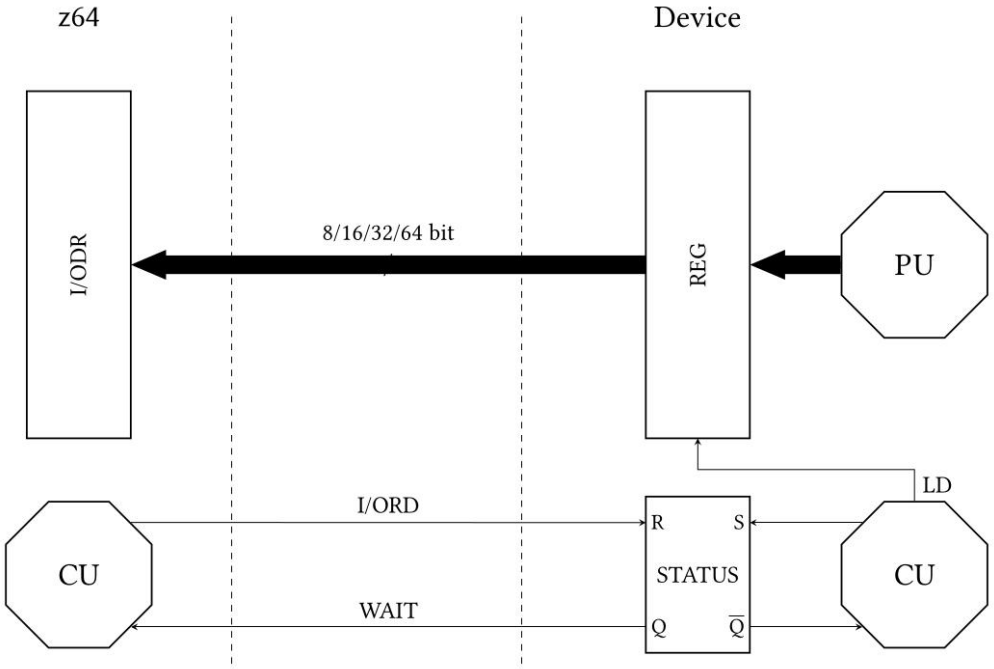
Potrei leggere prima che la scrittura sia avvenuta

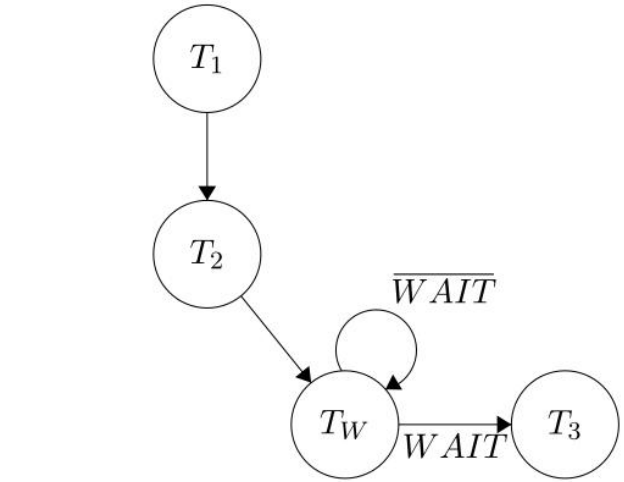
Cosa succede se si rimuovono le wait

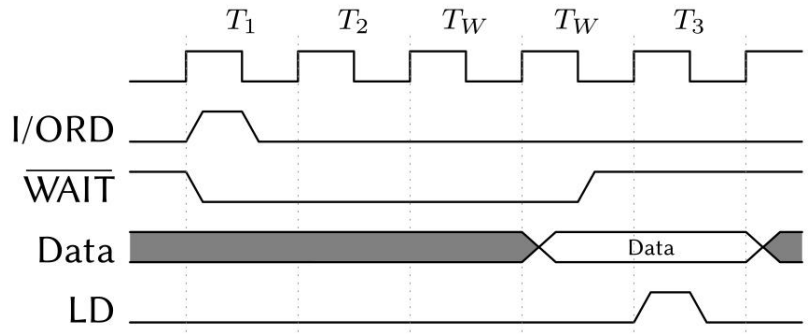


I/O microprogrammato per *un solo* dispositivo: input

- Interazione implementata a firmware

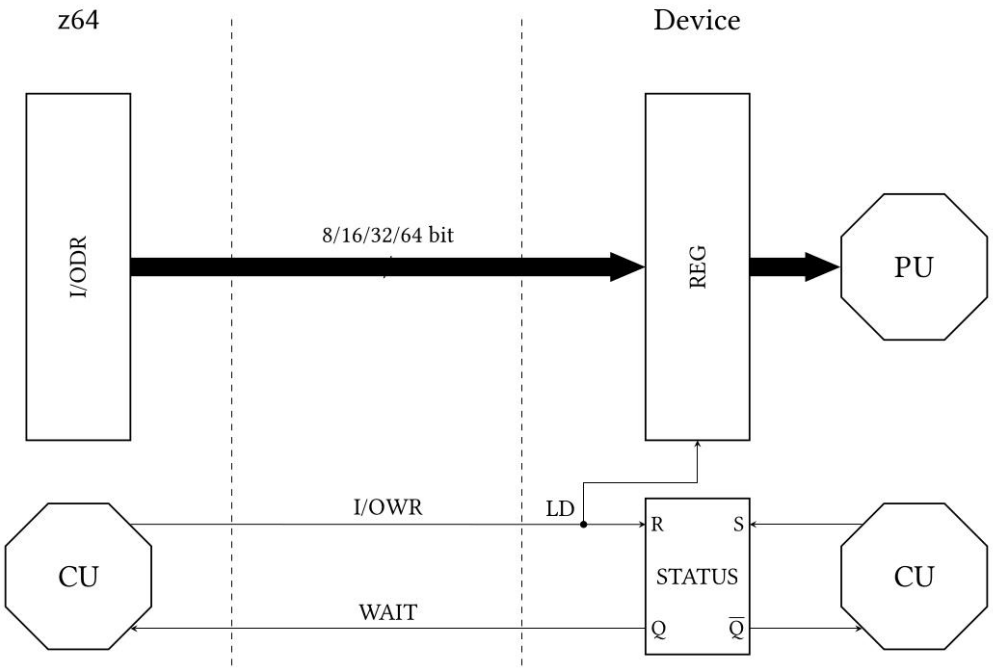


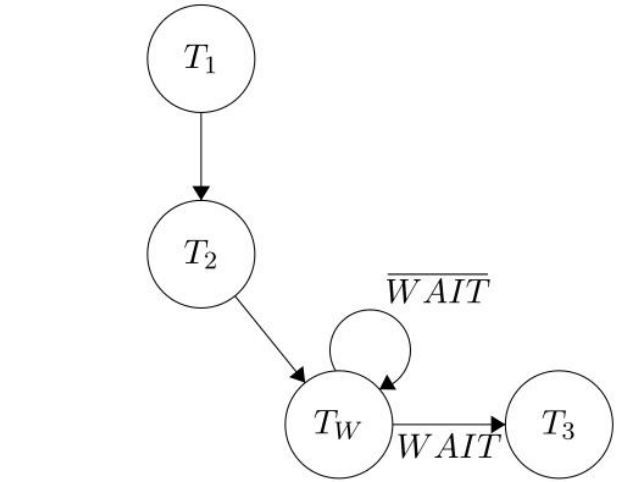


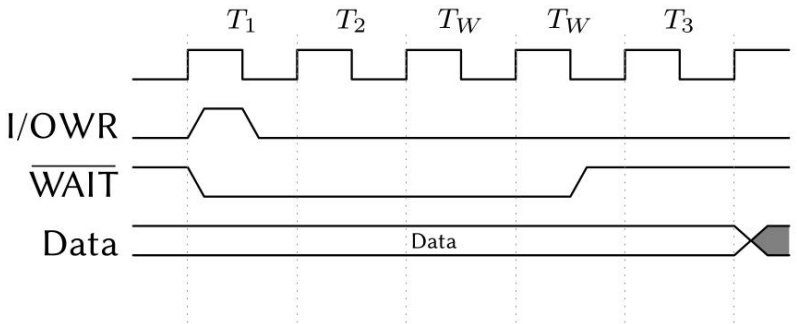


I/O microprogrammato per *un solo* dispositivo: output

- Interazione implementata a firmware

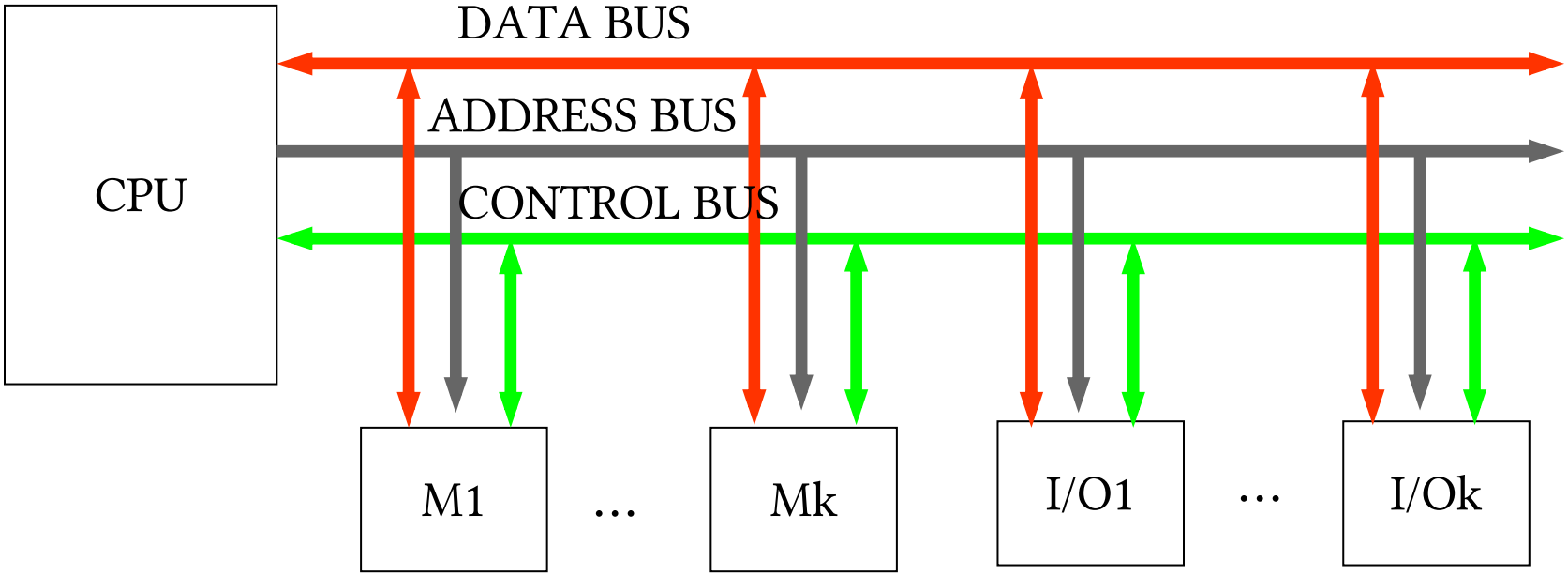






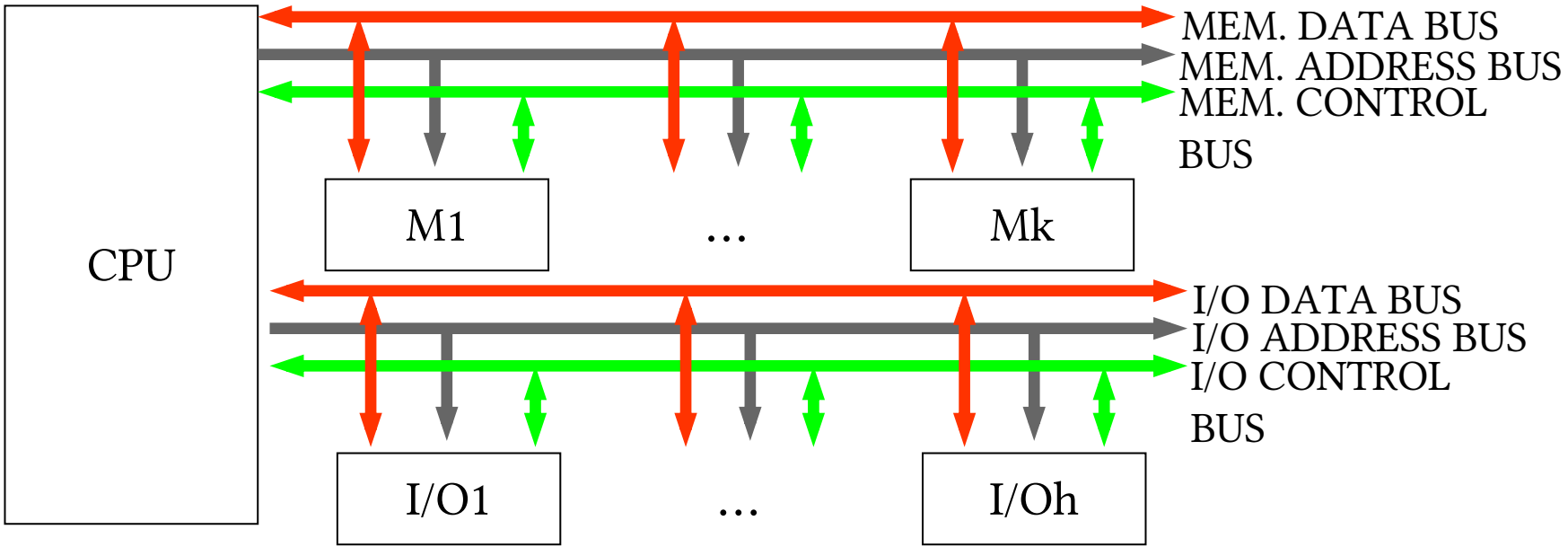
Possibili connessioni CPU-dispositivi

- Utilizzo di un singolo bus condiviso con la memoria
  - Condivisione dello spazio di indirizzamento tra memoria e dispositivi
  - *Memory-mapped I/O*: le istruzioni `mov` permettono il trasferimento dati



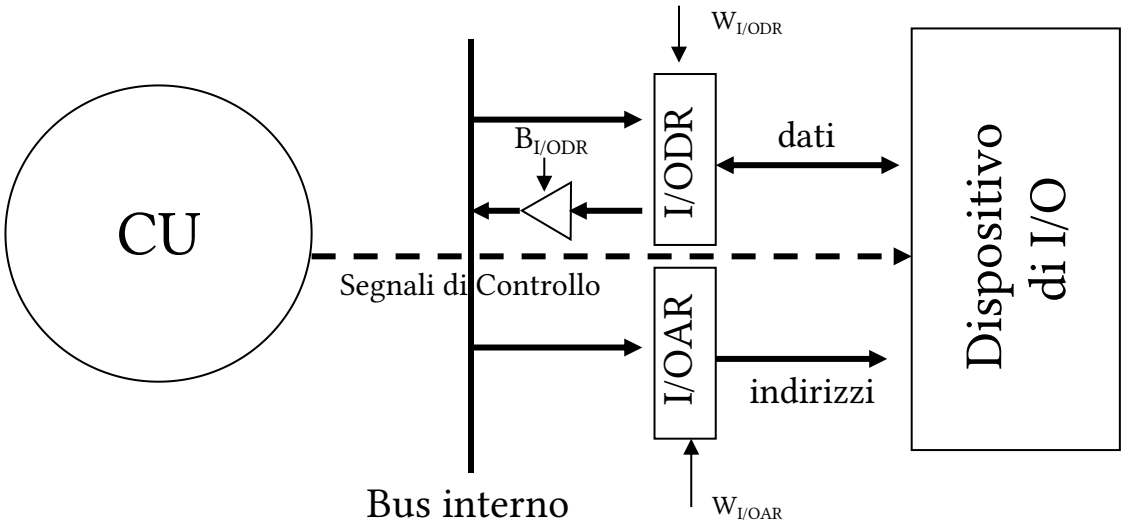
Possibili connessioni CPU-dispositivi

- Architettura a due bus: bus di memoria distinto dal bus di I/O
  - Spazi di indirizzamento separati
  - Necessità di istruzioni e registri di interfaccia dedicati

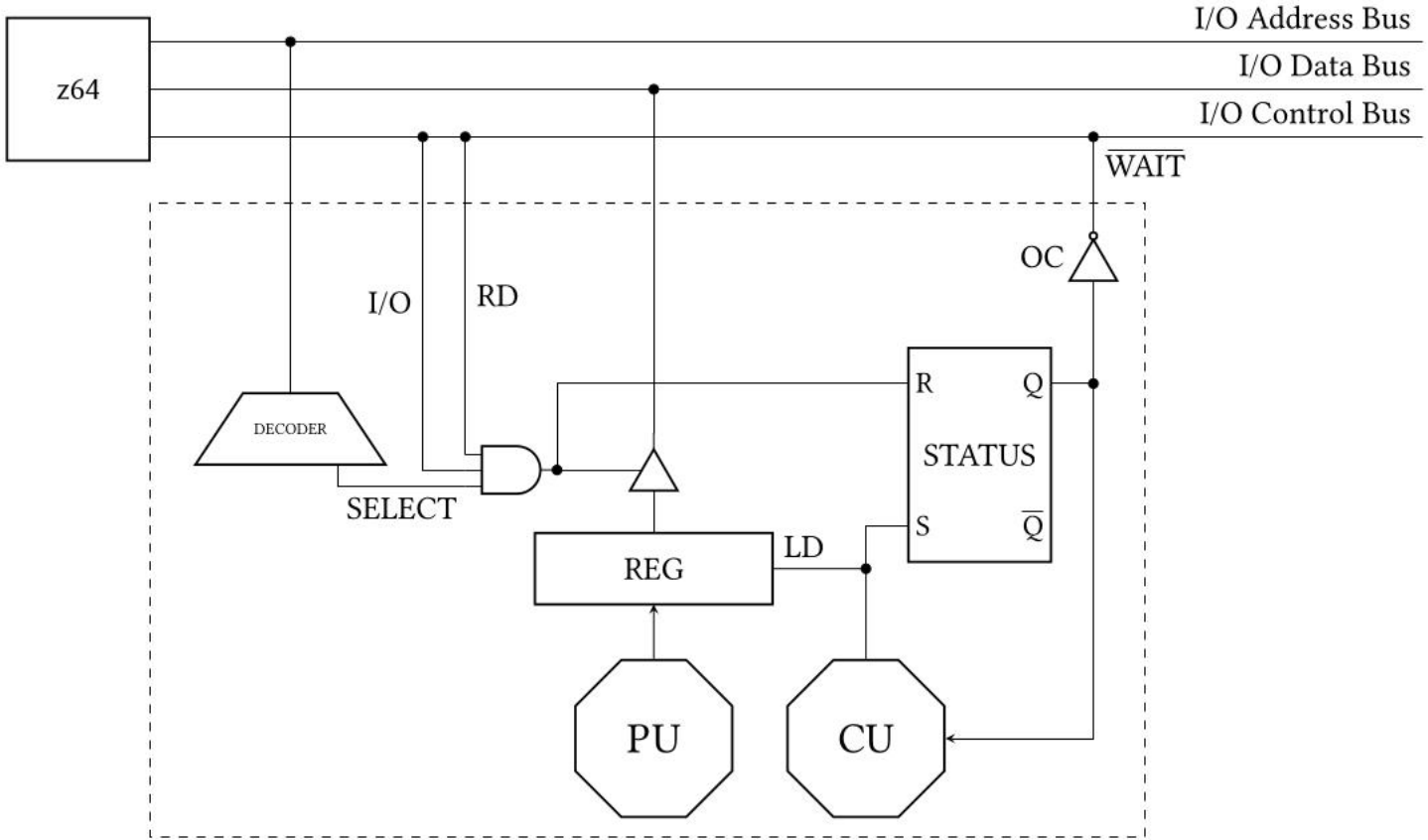


Interfaccia dello z64

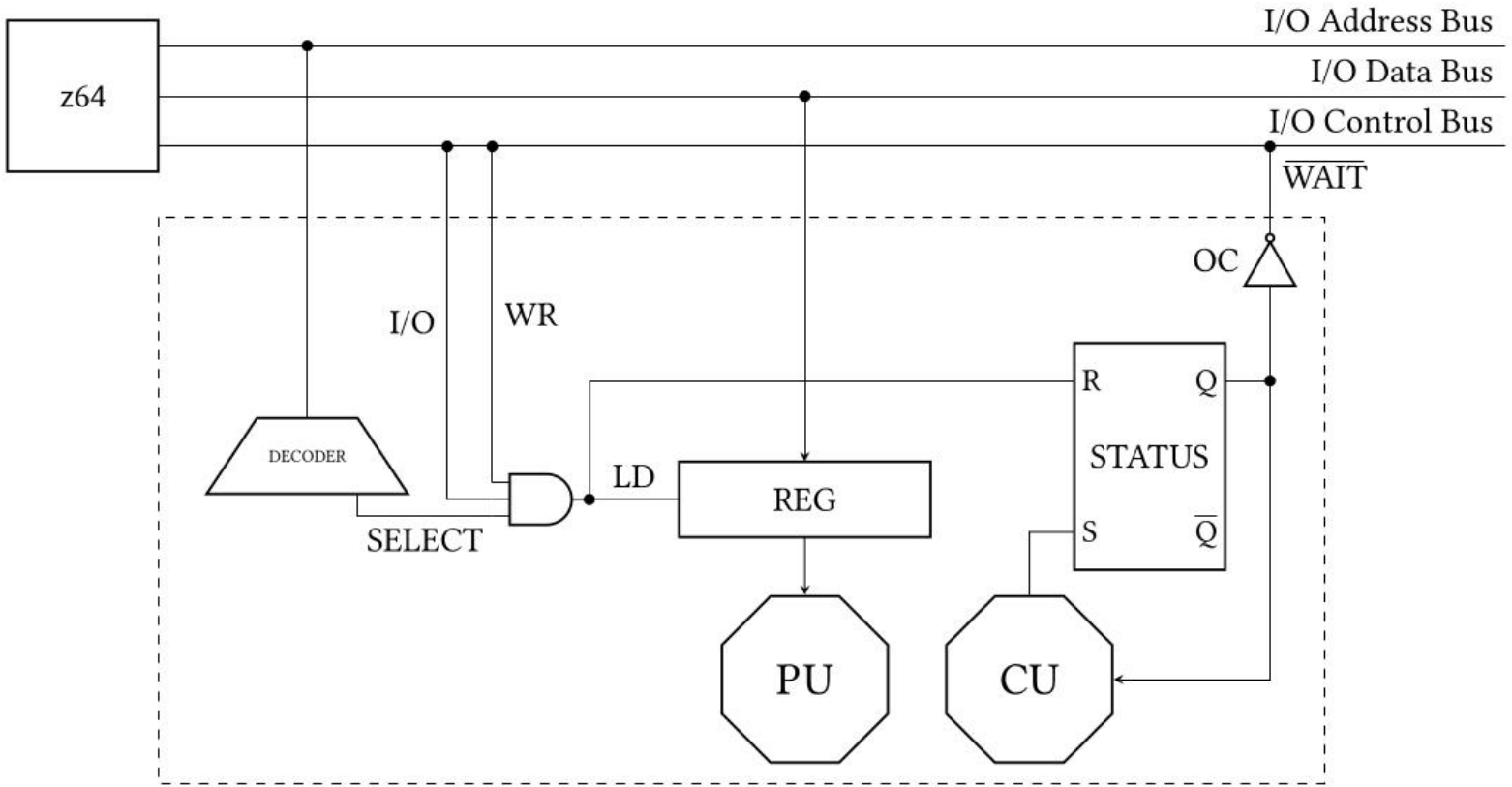
- Registro Dati (I/ODR)
- Registro Indirizzo (I/OAR)
- Segnali di Controllo (I/O, RD, WR, ...)



Interfaccia di input per connessione di più dispositivi



Interfaccia di output per connessione di più dispositivi



I/O gestito a software

- L'I/O microprogrammato manda in stallo il processore sul segnale di $\overline{\text{WAIT}}$ per un tempo possibilmente inaccettabile.
- Si può decidere di spostare il controllo delle operazioni di I/O al livello software, utilizzando tre principali strategie differenti.

- *I/O programmato*: è il programma software che inizializza e governa le interazioni con i dispositivi per effettuare il trasferimento di dati.
- *Su richiesta esterna*: il dispositivo richiede l'attenzione della CPU che esegue un *driver* per gestire il trasferimento dati.
- *Gestite da processori dedicati* (*canali* o *adattatori di bus*): il processore demanda ad un coprocessore l'operazione di trasferimento dati.

I/O programmato

- Il software guida lo scambio dei dati tra periferiche e CPU
- Possibile necessità di istruzioni dedicate: perché?
- Due modalità principali:
  - *busy waiting*
  - *polling*

Massimo 2^{16} *porte di I/O*

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`.

Busy Waiting

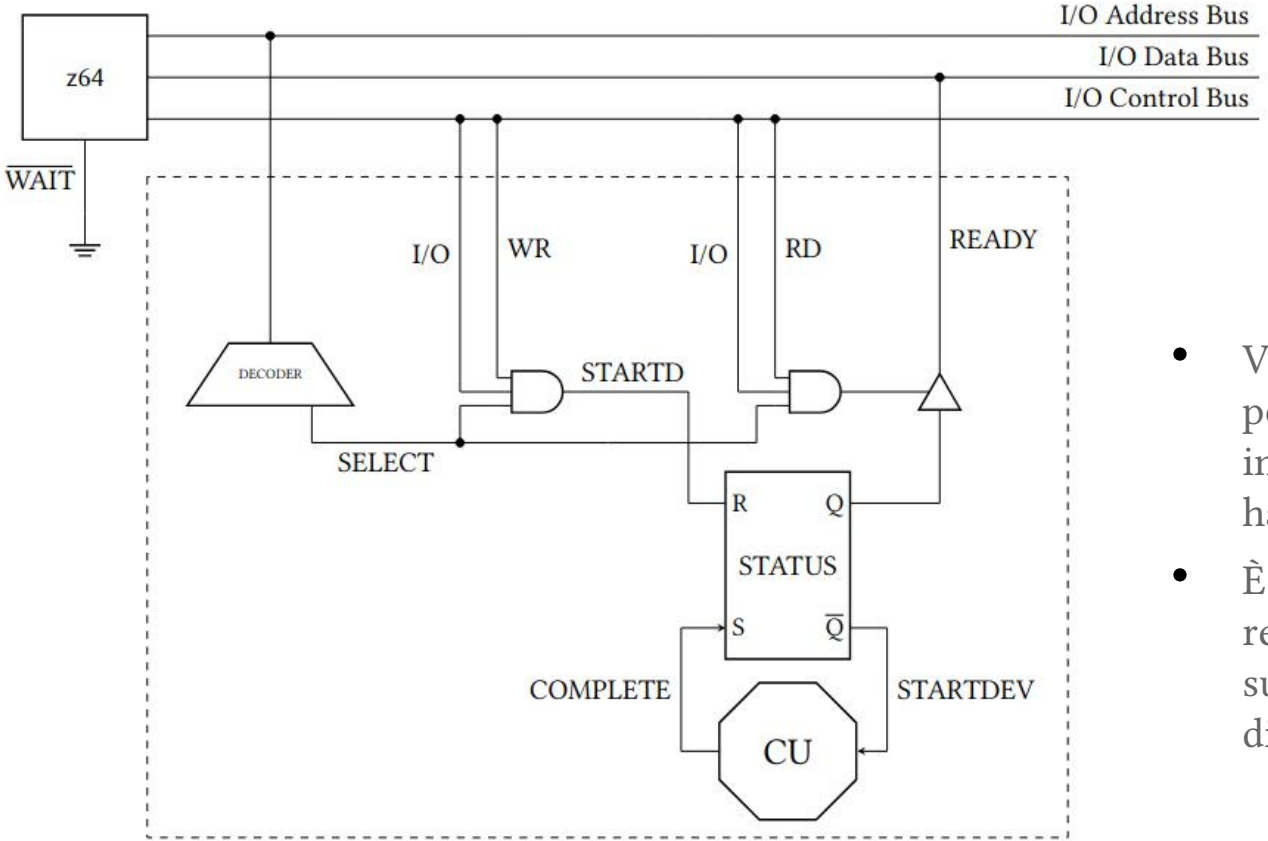
1. Il processore avvisa il dispositivo che vuole effettuare un trasferimento
2. Il processore verifica se il flip-flop STATUS è a 1
3. Se è a 0, il traferimento non è ancora completato: torna al punto 2
4. Il programma prosegue nel suo normale flusso

Busy Waiting

- Non vi è la possibilità che più dispositivi utilizzino il bus se non interrogati esplicitamente dalla CPU
- Non è più necessario il segnale $\overline{\text{WAIT}}$: collegato a massa

- La modalità busy waiting (*attesa attiva*) soffre della stessa problematica dell'I/O microprogrammato
- Il processore non esegue nessun'altra attività fino al completamento del trasferimento

Interfaccia per I/O programmato



- Viene mostrata solo la porzione di circuito per implementare il protocollo di handshaking
- È possibile aggiungere registri di interfaccia per supportare il trasferimento di dati in ingresso/uscita

Interazione busy waiting: software

```
        # %al è d.c.c.
        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

Polling

- Verifica circolare (simile al busy waiting) per trovare la prima periferica pronta ad interagire

```
.poll :
    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*

Interruzioni

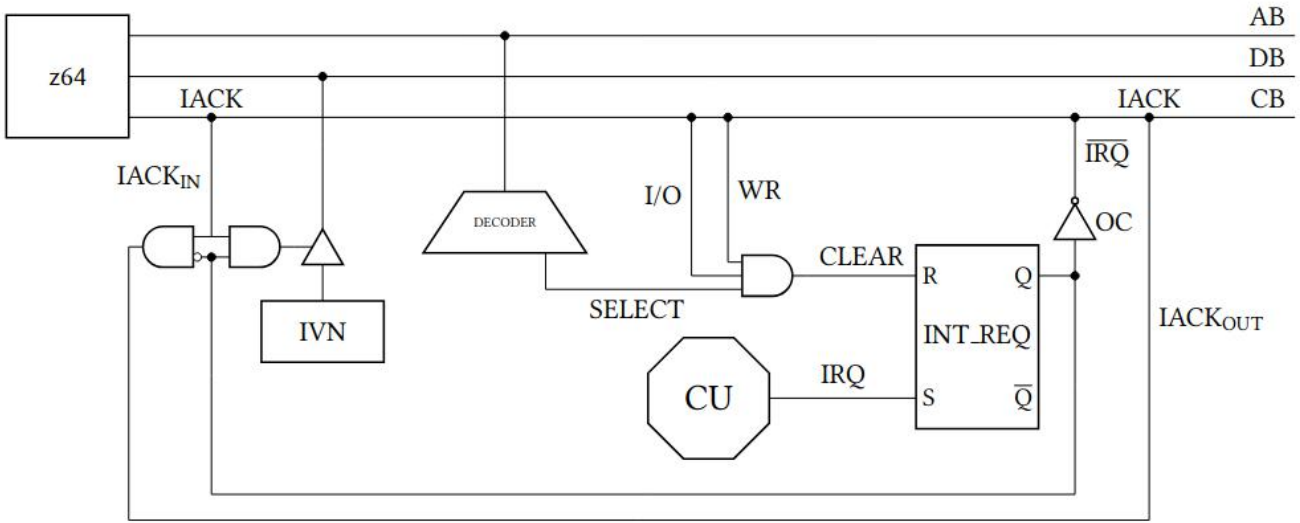
- Lo scopo delle interruzioni è quello di forzare il processore a *sospendere le attività correnti* per attivare l'esecuzione di un altro frammento di codice (*gestore* o *driver*)
  - La necessità dell'uso di driver nasce dall'eterogeneità di dispositivi che possono essere interconnessi alla CPU
  - Il progetto delle interfacce è "libero", a patto che si utilizzino correttamente i segnali di controlli da/verso la CPU
- I dispositivi, quando sono pronti ad interagire, *sollevano* una *richiesta di interruzione*
- Al termine dell'esecuzione del gestore, il processore riprende la normale esecuzione del programma

Gestione delle richieste di interruzione

- Ci sono vari problemi nella gestione delle richieste di interruzione
  1. La generazione di richieste di interruzione è un'attività *asincrona*
  2. Più dispositivi possono richiedere *contemporaneamente* l'attenzione della CPU
  3. Il processore può eseguire un solo driver per volta
  4. Il processore deve poter identificare quali dispositivi hanno sollevato la richiesta di interruzione (per identificare il driver)
  5. Il programma interrotto deve essere correttamente ripristinato al termine dell'esecuzione del gestore della richiesta di interruzione
- Soluzione al punto 1:
  - Il processore verifica periodicamente la presenza di un segnale di richiesta di interruzione (*interrupt request*, IRQ) proveniente dai dispositivi
- Soluzione ai punti 2, 3 e 4:
  - Dare una *priorità* alle richieste di interruzione (*daisy chain*)
  - Realizzare un protocollo (basato su firmware) per l'identificazione dei dispositivi che hanno sollevato richieste di interruzione

Interfaccia a daisy chain

- Tutti i dispositivi possono alzare il segnale $\overline{\text{IRQ}}$ (funzionante in open collector, quindi in logica negata)
- Il processore utilizza un segnale di *acknowledgement* per comunicare che è pronto a servire una richiesta di interruzione
- I dispositivi più vicini hanno priorità maggiore
- Identificazione basata su codice numerico (Interrupt Vector Number)



Interferenze con il programma interrotto

- Consideriamo il seguente frammento di codice eseguito su un processore multiciclo:

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

- Una richiesta di interruzione potrebbe essere ricevuta durante:
  - l'esecuzione del microprogramma dell'istruzione `movq`
  - tra l'istruzione `movq` e `testq`
- Il gestore dell'interruzione può contenere qualsiasi istruzione assembly, ad esempio `movq $1, %rax`
- Problemi:
  - I registri invisibili potrebbero essere sporcati
  - Il registro `FLAGS` potrebbe essere alterato
  - Il contenuto del registro RAX potrebbe essere stato modificato

Cambio di contesto (di esecuzione)

- Soluzione al punto 5: per attivare l'esecuzione di un gestore, il processore segue i seguenti passi:

1. Salvataggio dello stato del programma in esecuzione
2. Identificazione del programma di servizio relativo alla periferica che ha generato la richiesta di interruzione (driver)
3. Esecuzione del programma di servizio
4. Ripresa delle attività lasciate in sospeso

- L'attivazione di un gestore determina un *cambio di contesto (di esecuzione)*

Cambio di contesto (di esecuzione)

- Il cambio di contesto è l'insieme delle attività eseguite dalla CPU per interrompere il flusso d'esecuzione corrente ed attivare il gestore
- Il *contesto di esecuzione* viene salvato su stack (*interrupt frame*)
- Il contesto di esecuzione è composto da:
  - Registro RIP: contiene l'indirizzo dell'istruzione da cui riprendere l'esecuzione al termine della gestione dell'interrupt
  - Registro FLAGS: i bit di condizione potrebbero non essere ancora stati controllati dal programma interrotto
  - Registri invisibili al programmatore (es: TEMP1 e TEMP2), per supportare la ripresa dell'esecuzione di operazioni logico/aritmetiche
  - Registri general-purpose: per rendere trasparente l'esecuzione del driver

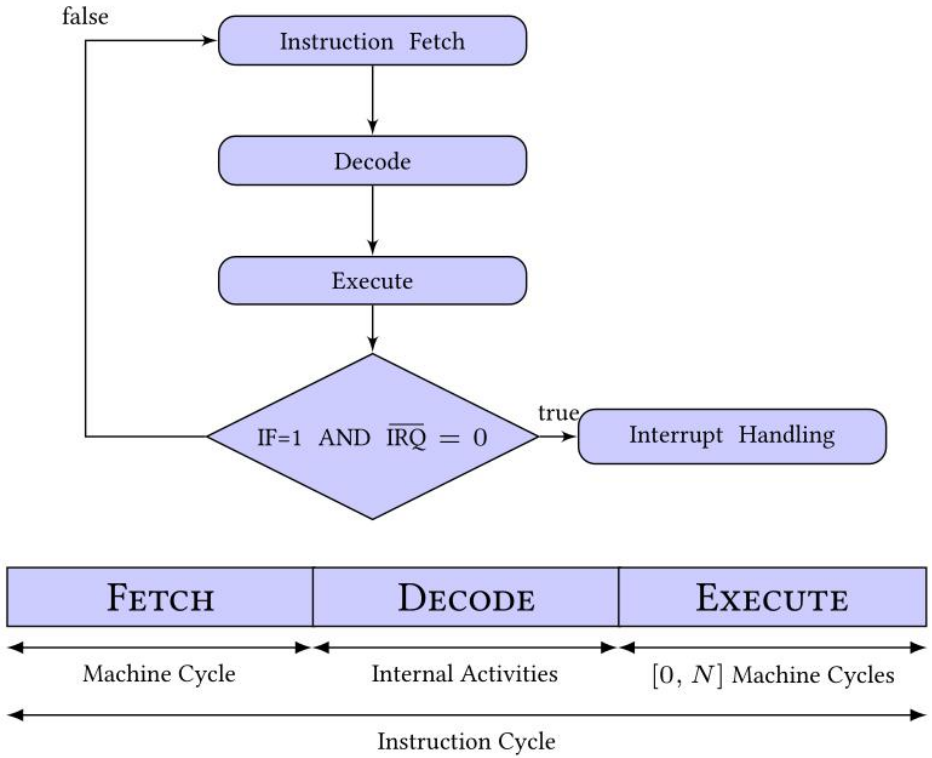
- Salvare tutti i registri può essere troppo costoso
  - Soluzione 1: il firmware salva solo alcuni registri, il software i restanti
  - Soluzione 2: differire il processamento di una richiesta di interruzione

Esecuzione atomica

- In alcuni casi, il programma in esecuzione deve poter essere *non interrompibile* (atomica = indivisibile)
  - Necessità di consentire un'esecuzione veloce e corretta di operazioni
  - Necessità di impedire la ricezione di altre richieste di interruzione durante il cambio di contesto

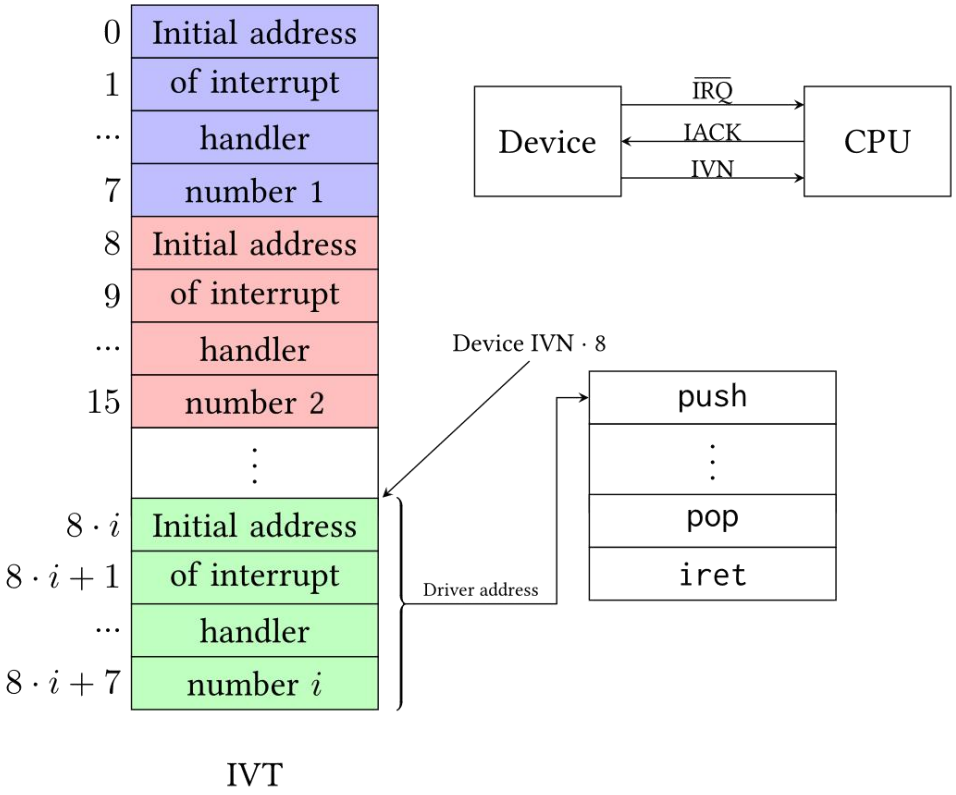
- Le richieste di interruzione devono essere *mascherabili*
- Si utilizza l'Interrupt Flag (IF) del registro FLAGS
  - se impostato a zero, le richieste di interruzione dei dispositivi vengono ignorate dalla CPU
  - Gestibile in maniera programmatica: `sti`/`cli`
  - Durante il cambio di contesto, IF viene azzerato dal firmware
- All'avvio del sistema IF=0 per permettere al software di registrare i driver necessari al funzionamento del sistema

Ciclo istruzione rivisitato



- Verificare la presenza di richieste di interruzione al termine del ciclo istruzione elimina la necessità di salvare il contesto completo (es, TEMP1 e TEMP2)
- RIP, FLAGS e i registri di uso generale devono comunque essere trattati

Associazione dispositivo/driver: interruzioni vettorizzate



- L'Interrupt Vector Table (IVT) permette di associare l'IVN ad un indirizzo (array di puntatori)
- A quell'indirizzo si trova la prima istruzione del driver
- La tabella deve essere popolata prima di abilitare le interruzioni

Microcodice per la gestione delle interruzioni

```
FLAGS[I] ← 0; TEMP1 ← RSP
TEMP2 ← 8
RSP ← ALU_OUT[SUB]
MAR ← RSP
MDR ← RIP
(MAR) ← MDR
TEMP1 ← RSP
TEMP2 ← 8
RSP ← ALU_OUT[SUB]
MDR ← FLAGS
MAR ← RSP
(MAR) ← MDR
IACK
IACK; MDR ← IVN
TEMP2 ← MDR
MAR ← SHIFTER_OUT[SX, 3] # only 256 different drivers!
MDR ← (MAR)
RIP ← MDR
```

- L'*interrupt frame* popolato dal firmware salva il contesto solo *parzialmente*
- I registri di uso generale non sono salvati
- Il driver deve salvare su stack *esplicitamente* tutti i registri utilizzati (sporcati)

Uscita dal contesto di interruzione: `iret`

```
MAR ← RSP
MDR ← (MAR)
FLAGS ← MDR
TEMP1 ← RSP
TEMP2 ← 8
RSP ← ALU OUT[ADD]
MAR ← RSP
MDR ← (MAR)
RIP ← MDR
TEMP1 ← RSP
TEMP2 ← 8
RSP ← ALU OUT[ADD]
FLAGS[I] ← 1
```

- Non è possibile utilizzare l'istruzione `ret` per concludere l'esecuzione di un driver
  - Lo stack frame è diverso dall'interrupt frame
- È necessario introdurre un'istruzione apposita per distruggere l'interrupt frame e restituire il controllo al programma interrotto

Driver di periferica

- L'esecuzione di un driver avviene con le interruzioni disabilitate
- È necessario rendere il codice del driver il più veloce possibile

- Tipica divisione in due parti
- *top half*: la parte di lavoro *non rinviabile*, eseguita con IF=0
  - recupero dei dati dal dispositivo (per evitare sovrascritture) se necessario
  - riprogrammazione della periferica (se necessario)
  - cancellazione della causa di interruzione (obbligatorio)
- *bottom half*: la parte di lavoro a priorità inferiore
  - es, processamento dei dati
  - può essere eseguita con le interruzioni riabilitate (esecuzione di `sti` esplicita)

Driver di periferica: esempio

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

Driver di periferica: esempio

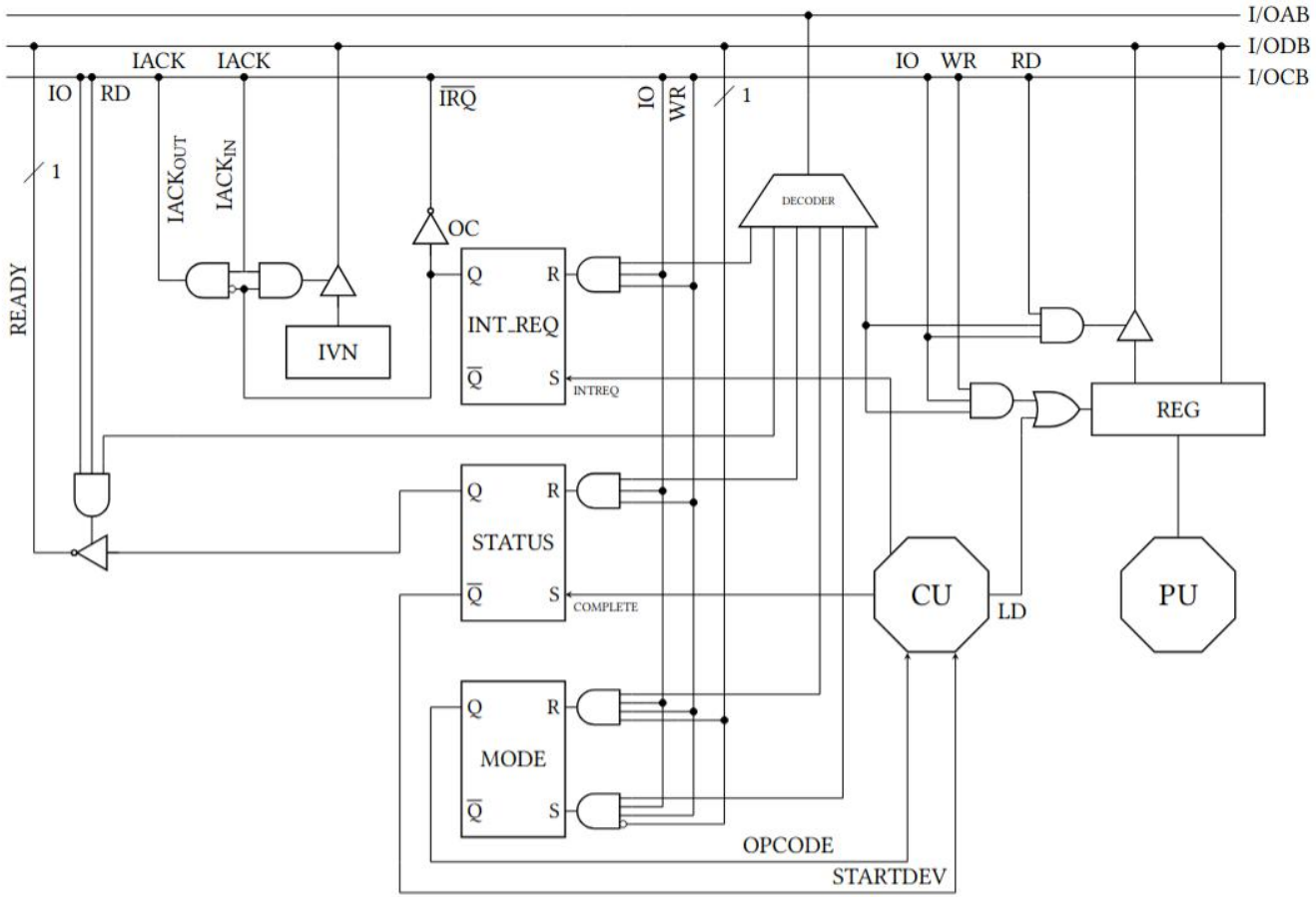
```
# restart the device to produce new data
# (if the device is supposed to be used like this)
outb %al, $device_status
# Destroy the interrupt frame and return control to the
# interrupted program
popq %rax
iret
```

Modalità mista e supporto di più operazioni

- Un singolo dispositivo può operare sia in modalità busy waiting che in modalità asincrona
  - Ad esempio, per fornire funzionalità di natura e con latenza differente
- È sempre il processore a determinare la modalità operativa del dispositivo
- Un singolo dispositivo può supportare l'esecuzione di operazioni differenti

- In entrambi i casi è possibile programmare il dispositivo utilizzando un registro/flip-flop di *opcode*

Modalità mista



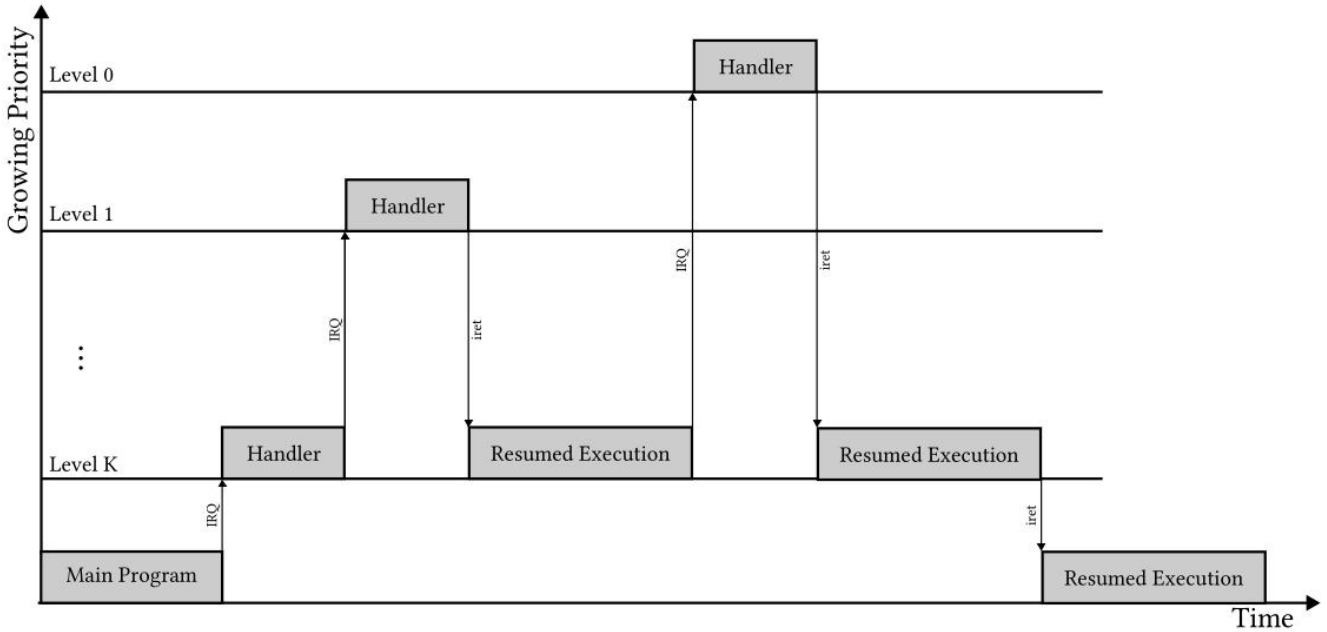
Gestione delle priorità

- Ci sono due problemi nella gestione delle priorità vista

1. Rischio di *priority inversion*
   - Se un driver non decide *esplicitamente* di essere interrompibile (usando l'istruzione `sti`), nessun dispositivo può interromperlo
2. Rischio di *starvation*
   - La richiesta di interruzione di un dispositivo a priorità minore potrebbe non essere mai servita a causa dell'organizzazione in daisy chain

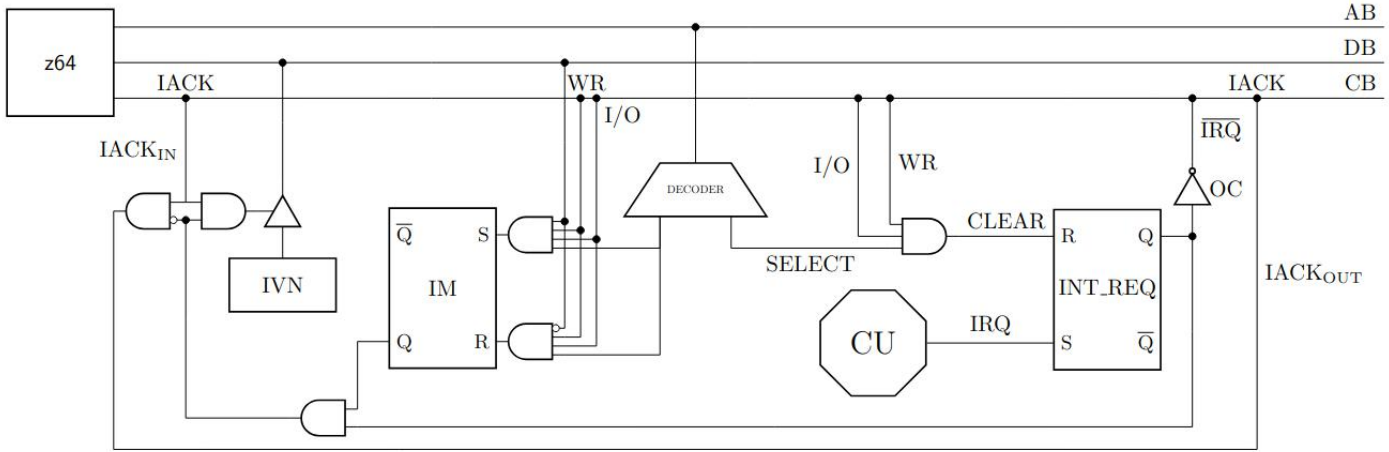
Gestione delle priorità

- Solo periferiche a priorità maggiore possono interrompere flussi d'esecuzione a priorità minore

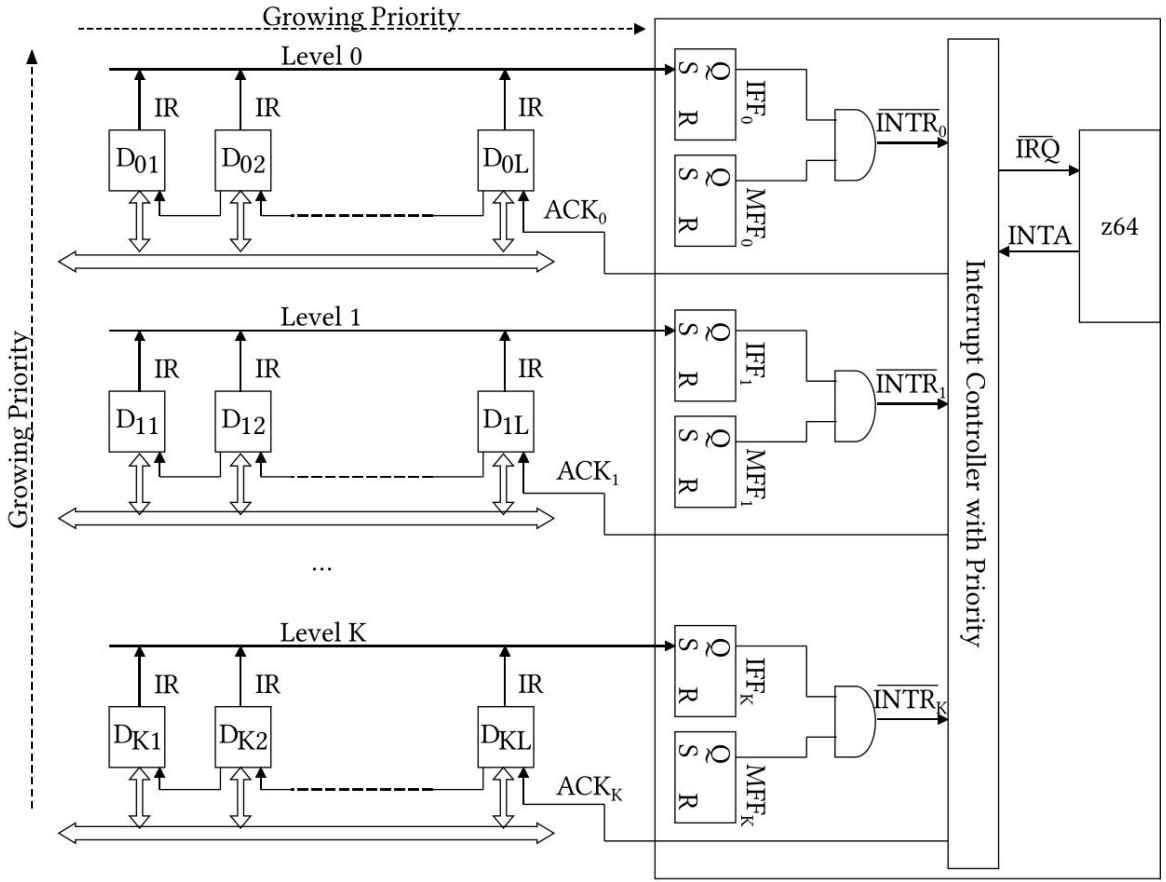


Gestione delle priorità a software

- Si può modificare l'interfaccia di un dispositivo per introdurre un F/F di *mascheramento delle interruzioni* che impedisce la generazione di richieste di interruzione
- Problematiche simili al polling:
  - priority inversion
  - starvation



Controllore degli interrupt a livelli di priorità



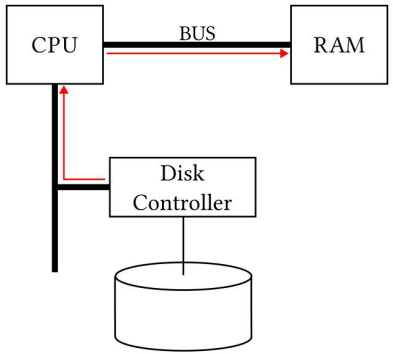
- Si possono utilizzare più daisy chain, ciascuna associata a un livello di priorità
- I flip-flop di maschera permettono di gestire i livelli di priorità

Interrupt non mascherabili

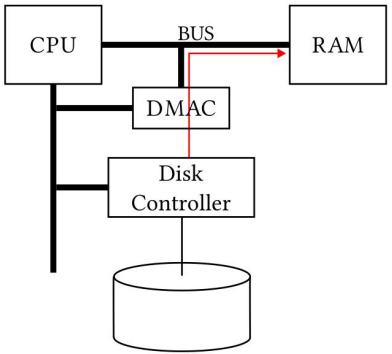
- Il mascheramento degli interrupt con il flag IF può causare il processamento ritardato di *attività critiche*
- Alcuni dispositivi a priorità elevatissima potrebbero non poter attendere che il processore termini le sue attività correnti per essere serviti
  - esempi: errori hardware non recuperabili, profilamento del sistema, reset del sistema, watchdog, batterie scariche
- Soluzione: aggiunta di una linea di *interrupt request non mascherabili* (Non-maskable Interrutp, NMI)
- I gestori dell'NMI devono essere velocissimi (tipicamente, non viene installato nemmeno l'interrupt frame)

Operazioni gestite da canale

- Nelle organizzazioni viste fino ad ora, il trasferimento dei dati è sempre mediato dal processore
  - Il trasferimento avviene alla grana dei tipi primitivi
- L'esecuzione di ciascuna istruzione `in`/`out` richiede un ciclo macchina per il fetch ed un ciclo macchina per il trasferimento
- Nel caso di trasferimenti di grandi moli di dati, l'operazione è lenta



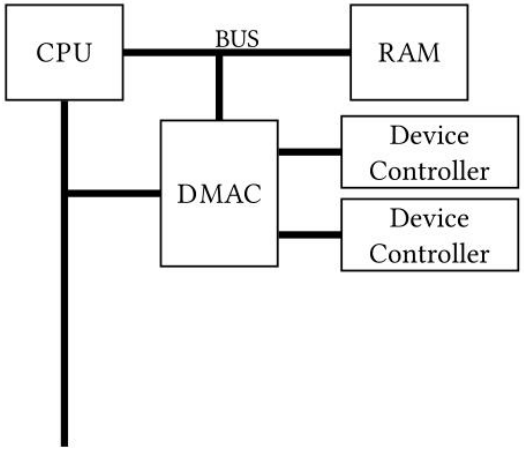
Accesso a memoria indiretto



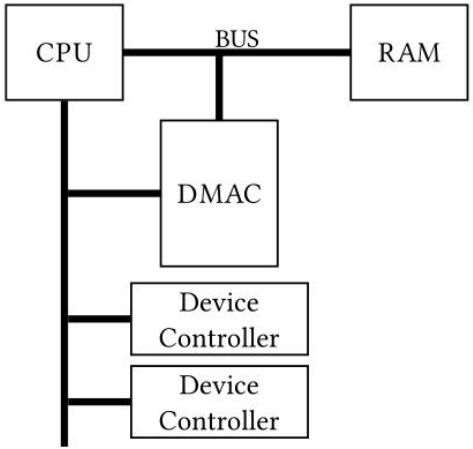
Accesso a memoria diretto

Organizzazione del DMAC

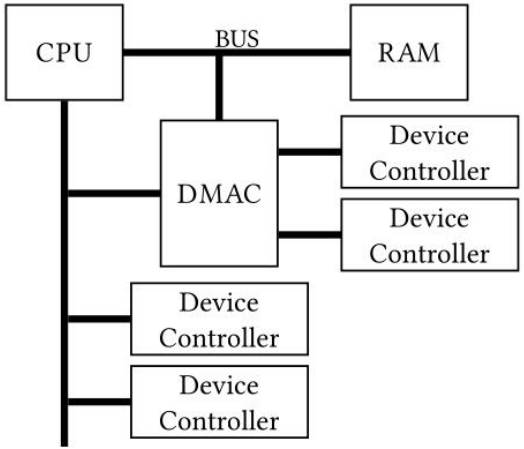
- Il Direct Memory Access Controller (DMAC) è un controllore di bus (coprocessore) che effettua trasferimenti tra dispositivi e memoria *al posto* della CPU
- Ci sono possibili organizzazioni differenti



Periferiche invisibili alla CPU



Periferiche condivise con la CPU



Modalità mista

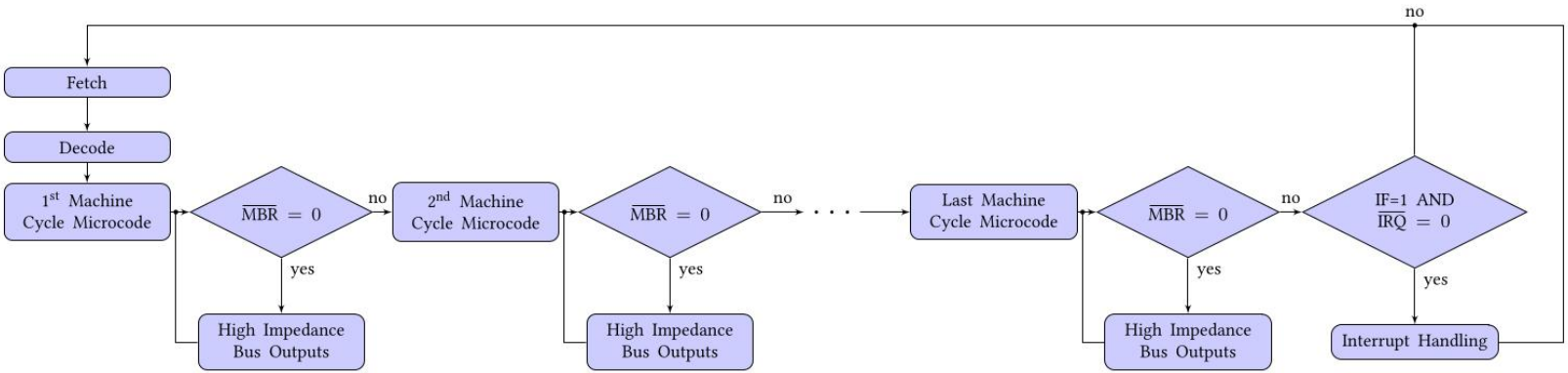
Condivisione del BUS

- In tutte le organizzazioni il DMAC e la CPU condividono l'accesso al bus di sistema
- È necessario *sincronizzare* le attività delle due componenti per evitare interferenze
- È necessario realizzare un protocollo di handshaking dedicato
  - Memory Bus Request ($\overline{\text{MBR}}$): allerta la CPU della volontà di effettuare un trasferimento dati da dispositivo a memoria
  - Memory Bus Grant (MBG): il processore rilascia il bus (alta impedenza) per permettere il trasferimento da parte del DMAC

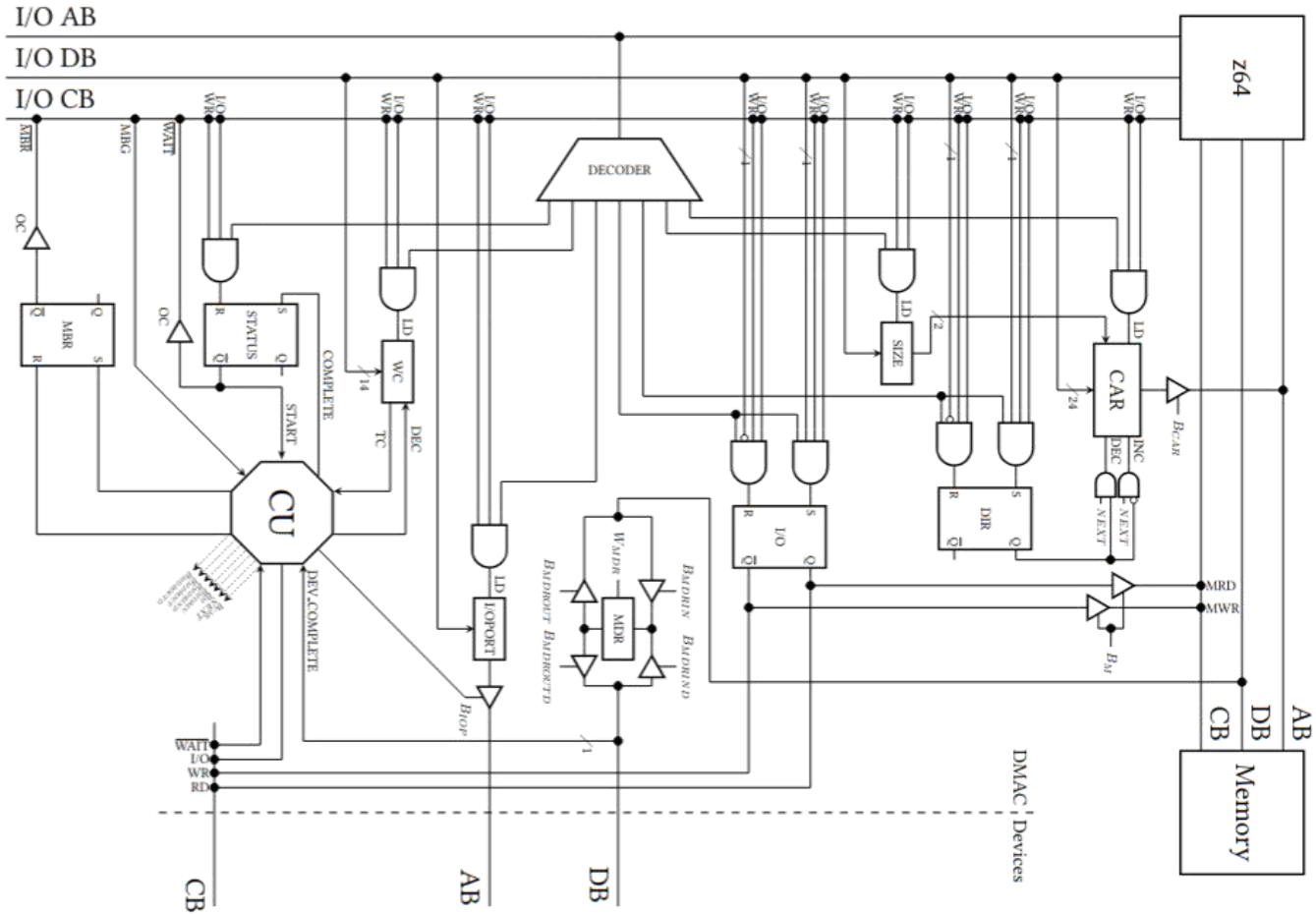
- Fintanto che il segnale $\overline{\text{MBR}}$ è asserito, la CPU è in stallo
- Cosa succede se il processore vuole effettuare un trasferimento dati mentre il DMAC sta trasferendo dati?

Protocolli di condivisione del BUS

- *Burst mode*: quando il processore asserisce MBG, il DMAC mantiene $\overline{\text{MBR}}$ fino al termine del batch di trasferimento
  - Sottoutilizzo del bus di sistema
  - Il DMAC passa la maggior parte del tempo ad interagire con le periferiche
  - Utile per dispositivi (come i dischi) che lavorano alla grana del blocco
- *Bus stealing*: il DMAC richiede il bus solo quando ha un dato pronto da trasferire



Interfaccia DMAC-processore



Programmazione del DMAC

- Il DMAC può essere programmato con delle istruzioni dedicate

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 $\overline{\text{WAIT}}$

Un'architettura moderna: adattatori di bus

- I dispositivi vengono organizzati per classi di *velocità*
- Dispositivi con velocità comparabili sono attestati su uno stesso bus
- I bus sono interconnessi tra loro tramite *adattatori di bus*
- Questi coprocessori dedicati implementano delle interfacce per effettuare trasferimenti a velocità differenti