DMA e PCI Bus Mastering

G. Lettieri

20 Maggio 2024

1 Direct Memory Access

Abbiamo visto, fino ad ora, due modalità di trasferimento dati da un dispositivo alla memoria, o viceversa:

- a "controllo di programma";
- tramite interruzioni.

Nella prima modalità il software controlla periodicamente che il dispositivo sia pronto leggendo un qualche registro di stato, quindi opera il trasferimento con una o più operazioni di lettura da registri del dispositivo, seguite da scritture in memoria, o viceversa. Nella modalità con interruzioni il trasferimento avviene nello stesso modo, ma è iniziato solo quando il dispositivo segnala la propria disponibilità tramite una richiesta di interruzione. La modalità a controllo di programma è più veloce di quella a interruzioni, ma, come sappiamo, è complicata da programmare se si vuole che il processore possa fare anche altro nei momenti in cui il dispositivo non è pronto; entrambe le modalità prevedono comunque un coinvolgimento del processore, che deve eseguire le istruzioni di lettura e scrittura, e comportano che i dati vengano scambiati sul bus due volte: una volta tra RAM e CPU e una volta tra CPU e dispositivo.

La modalità DMA (Direct Memory Access), che è la terza e ultima modalità di trasferimento dati, prevede invece che sia direttamente il dispositivo ad eseguire le necessarie operazioni di lettura o scrittura sulla RAM, una volta istruito dal software. In generale, il software vorrà eseguire un trasferimento dati dal dispositivo verso un buffer in RAM (operazione di ingresso o lettura) o da un buffer in RAM verso un dispositivo (operazione di uscita o scrittura). Supponiamo che il buffer si trovi all'indirizzo b e sia grande n byte, occupando dunque gli indirizzi $[b, b + n)$. L'indirizzo b e il numero n devono essere comunicati al dispositivo, insieme a tutte le altre eventuali informazioni necessarie a definire il trasferimento richiesto. Da quel momento in poi il dispositivo si preoccuperà di eseguire autonomamente le operazioni in RAM, al proprio ritmo. Ovviamente il dispositivo deve essere dotato di un sommatore che gli permetta di calcolare da solo gli indirizzi necessari a partire da b, e di un contatore che decrementi n ogni volta che è stato completato un trasferimento. Quando tutti i byte sono stati

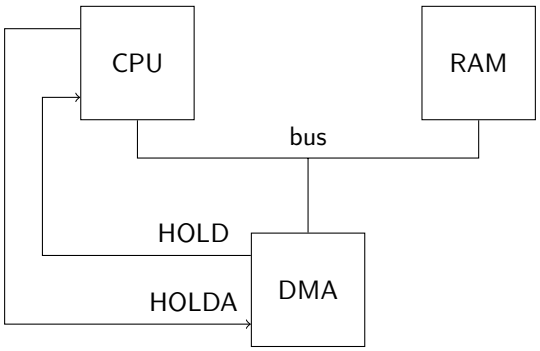

Figura 1: DMA, architettura di base.

trasferiti, il dispositivo segnalerà il completamento dell'operazione settando opportunamente un suo registro di stato e, tipicamente, inviando una richiesta di interruzione. Assumiamo che il software non acceda al buffer per tutto il tempo che intercorre tra l'avvio dell'operazione (operata dal software stesso) e la sua conclusione (segnalata dalla richiesta di interruzione).

Dal punto di vista hardware possiamo considerare l'architettura schematizzata in Figura 1, dove abbiamo per il momento eliminato cache, MMU e bus PCI, ricreando l'architettura tipica dei primi PC IBM, o dei minicomputer PDP della DEC. Il dispositivo in grado di operare in DMA è collegato direttamente al bus dove si trovano anche la CPU e la RAM. Questo gli permette di dialogare con la RAM usando lo stesso protocollo già usato dalla CPU. Dal momento che il bus è condiviso, però, un solo dispositivo alla volta può pilotarlo: o la CPU, o il dispositivo DMA. L'accesso è arbitrato tramite i collegamenti HOLD/HOLDA fra dispositivo e CPU:

1. il dispositivo mantiene normalmente i suoi piedini di uscita in alta impedenza;

2. ogni volta che il dispositivo vuole eseguire un trasferimento sul bus, attiva HOLD;

3. in risposta, la CPU termina l'eventuale trasferimento in corso (che può essere anche nel mezzo di una istruzione), mette i suoi piedini di uscita in alta impedenza e attiva HOLDA;

4. il dispositivo attiva i suoi piedini di uscita ed esegue il trasferimento, quindi rimette le uscite in alta impedenza e disattiva HOLD;

5. la CPU disattiva HOLDA, attiva i suoi piedini e riprende il suo normale funzionamento.

In pratica. la CPU dà la precedenza al DMA nell'accesso al bus. La tecnica è chiamata "cycle stealing", in quanto il DMA ruba cicli di bus alla CPU.

Attenzione a non confondere il protocollo HOLD/HOLDA con l'inizio e la fine dell'intera operazione di trasferimento: il dispositivo DMA avvierà il protocollo HOLD/HOLDA ogni volta che sarà pronto a trasferire m byte, ma in generale sarà $m \leq n$ e il protocollo andrà ripetuto più volte fino al completamento dell'intera operazione di trasferimento richiesta dal software. Negli intervalli di tempo in cui HOLD è disattivo la CPU può continuare a usare il bus indisturbata.

Si noti però che, mentre HOLD è attivo, la CPU può solo eseguire una istruzione già prelevata che non preveda accessi in memoria. Durante il tempo di trasferimento, dunque, la CPU potrebbe essere rallentata nell'esecuzione del software. Il meccanismo del DMA resta comunque vantaggioso in almeno tre scenari:

1. se la CPU è più lenta della RAM;
2. se il trasferimento a controllo di programma non è abbastanza veloce per il dispositivo;
3. se il dispositivo deve trasferire i dati con più urgenza di quanto permesso dal meccanismo delle interruzioni.

Il primo scenario era comune, per esempio, negli home computer degli '80 del secolo scorso, e il DMA era usato per trasferire dati di una schermata dalla RAM alla scheda video mentre la CPU stava eseguendo il programma che preparava la prossima schermata. Il secondo e terzo scenario possono verificarsi anche oggi, per esempio con alcune schede di rete che possono ricevere o inviare decine di milioni di pacchetti al secondo a velocità di 200 Gbps (Gigabit per secondo).

Il meccanismo HOLD/HOLDA funziona se c'è un unico dispositivo in grado di operare in DMA. Ci limitiamo a questo caso perché, come vedremo, il bus PCI ci permette di aggirare il problema senza introdurre altri meccanismi.

1.1 Interazione con la cache

Ora prendiamo in considerazione l'esistenza della cache. Notiamo subito che, come mostrato in Figura 2, i segnali HOLD e HOLDA devono ora collegare dispositivo DMA e controllore cache, in quanto è quest'ultimo, e non la CPU, ad essere collegato direttamente al bus con la memoria. A parte questo, l'handshake per il possesso del bus resta lo stesso e valgono le stesse considerazioni fatte precedentemente.

L'introduzione della cache porta un grande vantaggio, ma anche alcune complicazioni. Il vantaggio è che è molto più probabile che la CPU riesca ad eseguire istruzioni mentre è in corso una operazione di DMA, perché può trovare istruzioni e operandi in cache. Il rallentamento della CPU durante i trasferimenti in DMA può ora considerasi trascurabile.

Le complicazioni nascono dal fatto che le operazioni in DMA potrebbero coinvolgere parti di RAM che erano state precedentemente copiate in cache. Nel caso di cache con politica *write-back* la cache potrebbe anche contenere modifiche che non sono ancora state ricopiate in RAM. Esaminiamo i problemi,

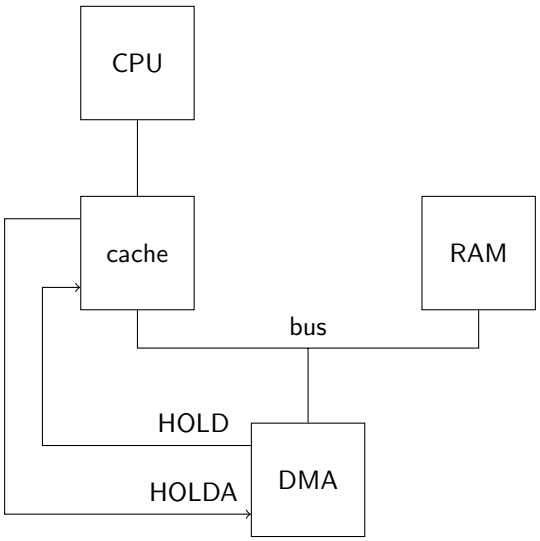

Figura 2: Interazione tra DMA e cache della CPU.

e le possibili soluzioni, partendo dal caso più semplice e passando poi a quelli più complessi. Ricordiamo che stiamo assumento che il software, per tutta la durata del trasferimento, non acceda al buffer coinvolto nel trasferimento. Ci interessa soltanto, dunque, lo stato della cache al momento di inizio e di fine del trasferimento.

1.1.1 Cache con politica *write-through*

Questo è il caso più semplice, perché all'inizio del trasferimento tutte le cacheline eventualmente presenti in cache contengono lo stesso valore delle corrispondenti cacheline in RAM. Questo implica che non ci sono problemi nel caso di operazione di uscita in DMA (direzione da RAM a dispositivo). Nel caso di operazione di ingresso in DMA (direzione da dispositivo a RAM), invece, bisogna assicurarsi che tutte le cacheline coinvolte (anche parzialmente) nel trasferimento vengano o rimosse dalla cache, o aggiornate, altrimenti il contenuto della cache, a fine trasferimento, non sarebbe consistente con il contenuto della RAM. Il software, dunque, potrebbe leggere valori "vecchi" scambiandoli per nuovi. Il problema è che l'accesso diretto alla memoria rende non più vera l'assunzione su cui abbiamo basato il funzionamento della cache, cioè che la RAM non cambia valore se non per le scritture che arrivano dalla CPU.

Il problema può essere risolto automaticamente in hardware, oppure essere demandato al programmatore software. Nel caso di soluzione completamente hardware, dobbiamo fare in modo che il controllore cache osservi *tutte* le possibili sorgenti di scritture in RAM. Grazie al bus condiviso, possiamo sfruttare il fatto che il controllore cache può osservare cosa sta accadendo durante l'accesso

diretto alla RAM, ricevendo in ingresso le linee di controllo e di indirizzo. Questo meccanismo viene chiamato *snooping* (ficcanasare), in quanto il controllore cache "ficca il naso" in quello che sta facendo il DMA. Se le linee di controllo identificano una operazione di scrittura, il controllore può usare il contenuto delle linee di indirizzo per eseguire una normale ricerca all'interno della cache (del tutto analoga a quella eseguita quando è la CPU a chiedere un indirizzo). Nel caso di *hit*, il controllore può autonomamente provvedere a invalidare la corrispondente cacheline. Alla fine del trasferimento, se il software accede al buffer, i dati dovranno essere prelevati dalla RAM, che contiene i valori aggiornati. Se il controllore riceve in ingresso anche le linee dati può aggiornare la cacheline, invece di invalidarla. Questa operazione viene detta *snarfing* (ingurgitare) e, tipicamente, non è prevista nei processori Intel. Si tenga presente che lo snarfing è più complicato di quanto può apparire a prima vista, in quanto il dispositivo DMA potrebbe star trasferendo byte che coinvolgono solo parte di una o più cacheline, non intere cacheline.

Ci sono sistemi (per esempio quelli basati su processori ARM, comuni negli smartphone) in cui il problema non è risolto automaticamente in hardware e il programmatore deve risolverlo in software. Per farlo dispone di alcune istruzioni che gli permettono di interagire con il controllore cache, tipicamente per invalidare un intervallo di indirizzi. Il software deve eseguire queste istruzioni, specificando l'intervallo di indirizzi coinvolto nel trasferimento (nel nostro caso, gli indirizzi da b a $b + n$ escluso) subito dopo che il trasferimento sia terminato. Per le cache write-through questa operazione di invalidazione è necessaria solo per i trasferimenti in ingresso.

1.1.2 Cache con politica *write-back*

Consideriamo ora una cache con politica *write-back* e ipotizziamo che il dispositivo DMA voglia leggere o scrivere dei byte di una cacheline attualmente presente in cache (senza perdita di generalità, limitiamoci a considerare una sola cacheline alla volta). Se la cacheline non è "dirty", la copia in cache contiene lo stesso valore che si trova nella corrispondente cacheline in RAM: ricadiamo negli stessi casi già visti per la cache di tipo write-through e possiamo adottare soluzioni analoghe. Se la cacheline è dirty, invece, ci scontriamo con dei problemi nuovi sia nelle operazioni di ingresso (da dispositivo a RAM) che nelle operazioni di uscita (da RAM a dispositivo):

- nelle operazioni di uscita, i dati più aggiornati si trovano in cache e se il DMA legge soltanto dalla RAM rischia di leggere valori vecchi;
- nelle operazioni di ingresso, il contenuto finale della RAM deve tenere conto sia dati contenuti in cache, sia di quelli che arrivano dal dispositivo DMA.

Si faccia attenzione al secondo punto, che è diverso dal problema che avevamo con la cache write-through: se la cache possiede una copia della cacheline in cui sta scrivendo il dispositivo DMA, e questa cacheline è marcata dirty, il

controllore cache non può, in generale, limitarsi a invalidare la propria copia. Infatti, ogni byte della RAM deve contenere l'ultimo valore scritto, sia se l'ultima scrittura proveniva dalla CPU, sia se proveniva dal DMA: se il dispositivo DMA non sovrascrive tutti i byte di una cacheline, gli altri byte devono continuare a memorizzare l'ultimo valore scritto dalla CPU, e questo valore si trova al momento soltanto in cache. Se il controllore cache invalidasse la propria copia senza eseguire prima un write-back, il contenuto più aggiornato di questi byte si perderebbe.

Le soluzioni hardware possibili sono diverse. Il problema in ingresso può risolversi con la tecnica di snooping adottata per le cache write-through, ma in questo caso il controllore *deve* implementare lo snarfing per le cacheline dirty. Il problema in uscita è più problematico. Infatti, è vero che il controllore cache può accorgersi tramite snooping che l'operazione di lettura dalla RAM iniziata dal dispositivo DMA coinvolge una cacheline dirty, ma cosa dovrebbe fare una volta scoperto ciò? In teoria dovrebbe essere il controllore a fornire i dati al dispositivo, invece della RAM, ma pilotare le linee dati con il valore aggiornato della cacheline comporta una corsa, in quanto anche la RAM sta vedendo la stessa operazione sul bus, e anche lei vorrà pilotare le linee dati. Occorre un coordinamento tra i tre attori coinvolti. Tipicamente il protocollo di accesso alla RAM viene modificato in modo che si svolga in più fasi temporalmente distinte:

1. c'è una prima fase, a cui la RAM non partecipa, in cui il dispositivo DMA comunica al controllore cache gli indirizzi a cui vuole accedere e il controllore cache risponde con il segnale hit/miss e l'eventuale stato del bit Dirty;

2. la prima fase è seguita da una o più fasi che dipendono dall'esito della prima.

La prima fase è in genere chiamata di snooping, ma in questo caso impropriamente, in quanto non si svolge "in segreto" come lo snooping di cui abbiamo parlato prima.

Consideriamo prima le operazioni di uscita. Se la fase di snooping è terminata con un miss o un hit non dirty, il dispositivo DMA può accedere normalmente alla RAM e il controllore cache non deve fare altro. Se la fase di snooping si è invece conclusa con un hit dirty si aprono varie possibilità, ma la più semplice è che il dispositivo DMA ceda il possesso del bus al controllore cache per permettergli di eseguire un write-back della cacheline in RAM; il dipositivo DMA può anche prelevare i dati che gli interessano mentre stanno transitando sul bus verso la RMA (in pratica eseguendo anche lui uno snooping con snarfing), altrimenti deve riacquisire il bus e rieseguire l'accesso quando il controllore cache ha terminato.

Se il controllore cache, come quelli Intel, non implementa lo snarfing, possiamo risolvere i problemi dei trasferimenti in ingresso in modo analogo a quello di uscita, prevedendo anche in questo caso una prima fase di snooping. Se la fase di snooping è terminata con miss o un hit non dirty, il dispositivo DMA può procedere con la normale scrittura in RAM e il controllore cache, in caso di

hit, deve invalidare la propria cacheline. Se la fase di snooping è terminata con un hit dirty, possiamo gestire il problema quasi esattamente come nel caso di ingresso: il dispositivo DMA cede il bus al controllore cache in modo da fargli eseguire il write-back in RAM, quindi riacquisisce il bus e riesegue la sua operazione di scrittura. A volte (come nel chipset emulato dalla nostra macchina QEMU) la soluzione è leggermente diversa: il controllore cache trasmette la cacheline dirty soltanto al dispositivo DMA (senza coinvolgere la RAM) e invalida la propria copia; il dispositivo DMA aggiorna la copia internamente con i dati che avrebbe dovuto scrivere in RAM, quindi scrive in RAM l'intera cacheline aggiornata.

Nel caso di soluzione interamente software, il programmatore può tipicamente ordinare al controllore cache di eseguire il write-back di un certo intervallo di indirizzi. Questa operazione è detta di *ripulitura*, e non comporta necessariamente anche una invalidazione. Il software deve eseguire questa operazione su tutti gli indirizzi del buffer, e deve farlo necessariamente *prima* di avviare il trasferimento. Questo è ovviamente necessario per le operazioni di uscita (da RAM a dispositivo), in modo che la RAM sia correttamente aggiornata prima del trasferimento, ma è necessario anche per le operazioni di ingresso (da dispositivo a RAM), non solo per preservare il valore attuale degli eventuali byte non coinvolti nel trasferimento, ma anche per evitare che eventuali cacheline dirty non vengano ricopiate in RAM durante l'operazione o dopo che è terminata, col rischio di sovrascrivere quanto già scritto dal dispositivo. Durante il trasferimento il software deve anche fare attenzione a non toccare eventuali byte che si trovino nelle stesse cacheline del buffer, pur non appartenendovi. La cosa più semplice per assicurarsi che ciò non avvenga è di avere buffer con dimensione multipla della cacheline e allineati alla cacheline (nel qual caso si può anche adottare l'ottimizzazione descritta di seguito).

1.1.3 Trasferimento di intere cacheline

Il caso di trasferimento in ingresso con cacheline dirty può essere molto semplificato se il dispositivo DMA ha intenzione di sovrascrivere l'intera cacheline e il controllore cache ne viene informato. In questo caso, il controllore cache può limitarsi a invalidare la propria copia come se non fosse dirty, e la fase di snooping preliminare non è necessaria. Per questo scopo, il protocollo del bus prevede in genere una operazione di "Write and Invalidate" distinta dalla normale operazione di scrittura. Il dispositivo DMA può usare questa operazione ogni volta che è sicuro di star sovrascrivendo una intera cacheline. In generale, un intero trasferimento che coinvolge un buffer $[b, b+n)$ sarà composto, in base all'allineamento di b e al valore di n, da un possibile trasferimento di parte di una cacheline, seguito da zero o più trasferimenti di intere cacheline, e concluso da un eventuale trasferimento di parte di una cacheline. Il dispositivo DMA potrà usare normali scritture (con fase di snooping) soltanto per la prima e l'ultima operazione, se necessario, e usare Write and Invalidate per quelle intermedie.

Nel caso di soluzione puramente software, il programmatore deve invocare l'operazione di ripulitura all'inizio del trasferimento, per evitare il write back di

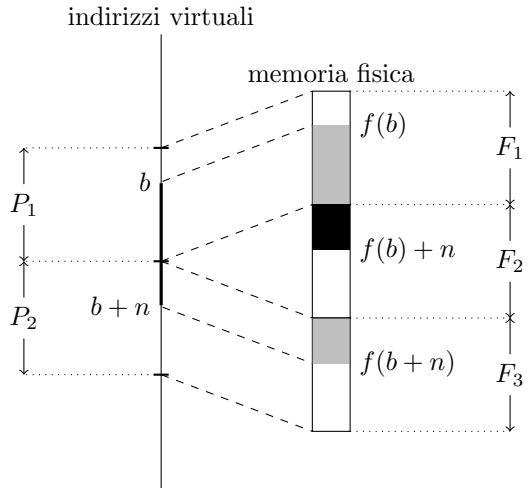

Figura 3: Un esempio con un buffer $[b, b+n)$ che attraversa due pagine (P_1 e P_2) mappate su due frame non contigui (F_1 e F_3).

eventuali cacheline dirty durante o dopo l'operazione DMA.

1.2 Interazione con la memoria virtuale

Come sappiamo, tra la CPU e la cache troviamo la MMU, che traduce gli indirizzi virtuali (usati dal software) in indirizzi fisici (usati sul bus). Il dispositivo DMA, come collegato in Figura 2, non interagisce in alcun modo con la MMU e può utilizzare soltanto indirizzi fisici. Supponiamo che il software voglia eseguire un trasferimento in DMA da un buffer che si trova agli indirizzi *virtuali* $[b, b+n)$ verso un dispositivo (o viceversa). Saranno necessari i seguenti accorgimenti:

- al dispositivo andrà comunicato l'indirizzo *fisico* corrispondente a b, sia $f(b)$, e non l'indirizzo virtuale b;
- se l'intervallo $[b, b+n)$ attraversa più pagine che non sono tradotte in frame contigui, il trasferimento deve essere spezzato in più trasferimenti in modo che ciascuno di essi coinvolga solo frame contigui;
- la traduzione degli indirizzi coinvolti in un trasferimento non deve cambiare mentre il trasferimento è in corso.

Tutti e tre i punti sono diretta conseguenza del fatto che il dispositivo non ha accesso alla traduzione da indirizzi virtuali a fisici. Consideriamo il primo punto: se comunicassimo b al dispositivo, questo lo userebbe come se fosse un indirizzo fisico, andando a leggere o scrivere in parti della memoria che non c'entrano niente con il buffer (tranne nel caso particolare in cui $f(b) = b$). Consideriamo il secondo punto e supponiamo che $[b, b+n)$ attraversi due pagine, P_1 e

P_2, mappate su due frame non contigui, F_1 e F_3 (si veda la Figura 3). Se al dispositivo comunichiamo $f(b)$ ed n, questo leggerà (o scriverà) agli indirizzi *fisici* $\big[f(b), f(b)+n\big)$, invadendo dunque il frame F_2, con effetti disastrosi. Il trasferimento, invece, deve essere spezzato in due parti: una da $f(b)$ fino alla fine di F_1, e uno dall'inizio di F_3 fino a $f(b+n)$ escluso. Alcuni dispositivi possono essere programmati per eseguire autonomamente più trasferimenti in successione, specificando in anticipo l'indirizzo e il numero di byte di ciascun trasferimento. Anche nei dispositivi che non offrono questa funzione, comunque, è sempre possibile programmare i diversi trasferimenti uno dopo l'altro in software.

Consideriamo invece il terzo punto, immaginando di trovarci in un sistema multiprocesso che realizzi, per esempio, lo swap-in/swap-out dei processi per poter eseguire più processi di quanti ne possano entrare in RAM. Supponiamo che un processo P_1 avvii un trasferimento in DMA verso un suo buffer privato e, mentre il trasferimento è in corso, il sistema decida di eseguire lo swap-out di P_1 per caricare un altro processo P_2 al suo posto. Il dispositivo DMA è ignaro del cambiamento e continuerà leggere o scrivere agli indirizzi fisici precedentemente occupati dal buffer di P_1 e ora occupati da parti della memoria di P_2, di nuovo con effetti disastrosi.

2 PCI Bus Mastering

La Figura 4 mostra un esempio di architettura con bus PCI. Come sappiamo, sul bus PCI ogni dispositivo (più precisamente, ogni funzione all'interno di ogni dispositivo) può comportarsi da "iniziatore" di una transazione. I dispositivi che sono in grado di essere iniziatori di transazioni sono detti *bus master*, e devono essere dotati dei collegamenti REQ/GNT verso l'arbitro, che ha il compito di coordinare l'accesso al bus tra i vari bus master. In Figura 4 abbiamo tre dispositivi collegati al bus PCI: il ponte, che è sempre bus master (in quanto deve iniziare le transazioni PCI per conto della CPU), un altro dispositivo bus master, e un terzo dispositivo che non è bus master. Solo il ponte e il secondo dispositivo hanno i collegamenti REQ/GNT con l'arbitro. Per ottimizzare i tempi, l'arbitraggio può svolgersi mentre è in corso una precedente transazione: il dispositivo che ottiene il GNT, prima di iniziare la propria transazione, deve attendere che il bus diventi libero. La condizione di bus libero si riconosce dal fatto che sia FRAME# che IRDY# sono disattivi.

Dal punto di vista software l'operazione si svolge come se il dispositivo bus master fosse collegato direttamente al bus principale; in particolare, il software dialoga soltanto con il dispositivo per comunicargli, per esempio, l'indirizzo (fisico) del buffer, il numero di byte da trasferire e la direzione del trasferimento. Dal punto di vista hardware, però, tutto il trasferimento avviene per tramite del ponte. Ogni volta che il dispositivo vuole iniziare un trasferimento da o verso la RAM, inizia una transazione di memoria sul bus PCI, all'indirizzo opportuno. Il ponte è configurato in modo da comportarsi come obiettivo per tutte le transazioni di memoria con indirizzi appartenenti alla RAM che si trova sul bus

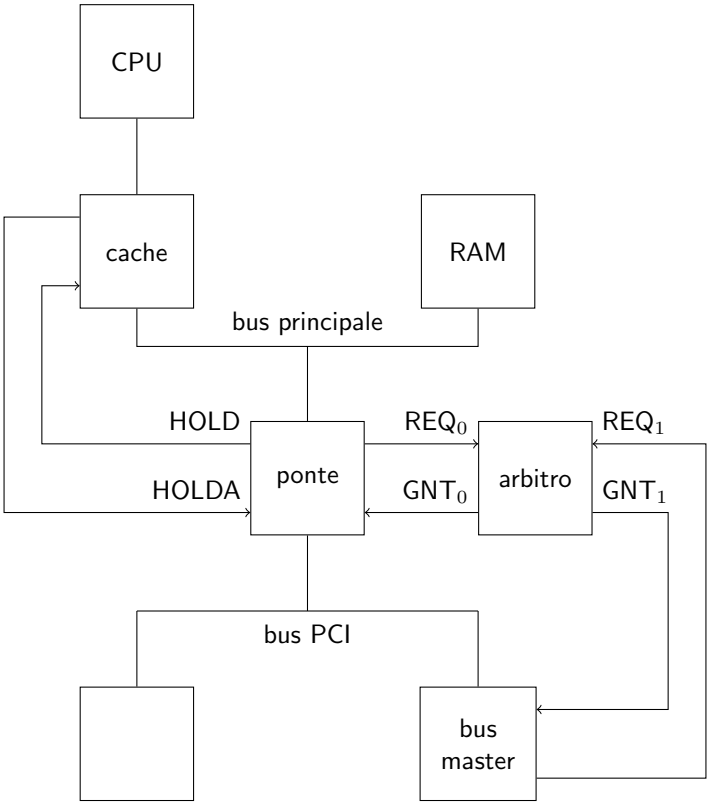

Figura 4: Bus Mastering PCI.

principale. Il ponte, quindi, risponde alla transazione iniziata dal bus master e, in caso di trasferimento da RAM a dispositivo, esegue le necessarie operazioni di lettura DMA sul bus principale; in caso di trasferimento dal dispositivo alla RAM, copia temporaneamente i dati in arrivo dal dispositivo in un buffer interno (*posted writes*) e, in parallelo, inizia le necessarie operazioni di scrittura in DMA verso la RAM. Sul bus principale il ponte si comporta esattamente come un normale dispositivo DMA e per lui valgono tutte le considerazioni che abbiamo svolto fino ad ora, in particolare per quanto riguarda il protocollo HOLD/HOLDA e le interazioni con la cache.

Si noti che il bus master può utilmente sfruttare la possibilità, offerta dallo standard PCI, di eseguire transazioni con più di una fase dati, in quanto conosce in anticipo il numero di byte da trasferire. Inoltre, anche il bus PCI prevede una transazione di tipo "Write and Invalidate" che può essere sfruttata dai dispositivi bus master per comunicare la loro intenzione di sovrascrivere intere cacheline. Lo standar PCI, però, richiede che il software comunichi ai dispositivi bus master la dimensione delle cacheline (si veda il campo "Cache Line Size" nello spazio di configurazione, Figura 4 della dispensa sul PCI); questo perché lo standard PCI non è legato ad un particolare modello di processore. Questo campo viene in genere scritto dal PCI BIOS durante la fase di inizializzazione del sistema.

2.1 Interazione con il meccanismo delle interruzioni

La presenza della bufferizzazione intermedia nel ponte crea un problema con i trasferimenti in ingresso (da dispositivo a RAM) e il meccanismo delle interruzioni. In particolare, quando il dispositivo ha trasmesso al ponte l'ultimo byte richiesto, invierà una richiesta di interruzione per segnalare il completamento del trasferimento. Ci troviamo a questo punto di fronte ad una corsa: da una parte la richiesta di interruzione, che passa dal controllore delle interruzioni e arriva alla CPU, dall'altra i byte destinati alla memoria, che passano dal ponte: è possibile che la richiesta di interruzione arrivi e sia accettata prima che il ponte sia riuscito a trasferire tutti i byte in RAM; il software potrebbe quindi accedere erroneamente al buffer quando ancora parte dei dati non sono stati aggiornati.

Anche questo problema può essere risolto in hardware o in software. Per risolverlo in software si può sfruttare il fatto che il ponte gestisce in modo strettamente FIFO tutti i trasferimenti di dati dal bus PCI verso il bus principale. La routine di interruzione, allora, prima di permettere l'acesso al buffer, potrebbe eseguire una lettura di un registro del dispositivo bus master. La risposta a questa lettura verrebbe accodata nel ponte *dopo* l'ultimo trasferimento di dati del bus master, e dunque l'istruzione di lettura verrebbe completata nella CPU necessariamente dopo che il ponte ha finito di trasferire i dati in RAM. Si noti che, dal momento che il dispositivo bus master aveva inviato una richiesta di interruzione, è probabilmente già previsto che la routine di interruzione debba leggere un qualche registro del dispositivo, per segnalare che la richiesta è stata servita: in tal caso, quest'unica lettura assolve entrambi i compiti.

Per risolvere il problema interamente in hardware, invece, si crea in genere un collegamento di handshake tra il controllore delle interruzioni e il ponte. Prima di inoltrare una qualunque richiesta di interruzione al processore, il controllore chiede l'OK al ponte; quest'ultimo non dà l'OK fino a quando non ha finito di trasferire tutti i dati ricevuti fino a quel momento. Questa è la soluzione adottata nel ponte che è emulato all'interno della nostra macchina QEMU.

Un'altra soluzione, più moderna, prevede che le richieste di interruzione non viaggiono su linee separate, ma siano inoltrate come speciali transazioni sul bus PCI stesso, sotto forma di scritture a particolari indirizzi (Message Signaled Interrupts). In questo modo le richieste si accodano naturalmente ai dati e non ci sono problemi di corse.