LABORATORIO DI ARCHITETTURA DEI CALCOLATORI

lezione n° 20

Prof. Rosario Cerbone

rosario.cerbone@libero.it

http://digilander.libero.it/rosario.cerbone

a.a. 2005-2006

Arbitrazione di più Masters

- La trattazione dei bus fino a questo punto partiva dal presupposto che nel sistema ci fosse un solo dispositivo in grado di agire come Master della comunicazione, e quindi di prendere l'iniziativa di iniziare un ciclo di lettura o scrittura inviando gli opportuni segnali CS e R/W allo Slave da lui individuato. Nella stragrande maggioranza dei sistemi, invece, ci si trova a dover trattare casi più complessi, nei quali anche altri dispositivi, oltre alla CPU, possono agire da Master nei confronti del bus di sistema. L'esempio classico é quello dei trasferimenti DMA nei quali un controller agisce per conto della CPU per trasferire dati da RAM ad altre unità (per esempio dischi) o viceversa.

Arbitrazione di più Masters

- La complicazione del caso in cui due o più dispositivi diversi sono predisposti per comportarsi come Master verso il bus deriva dal fatto che ciascuno di questi potrebbe prendere l'iniziativa di iniziare un ciclo di lettura o scrittura ed occorre prendere dei provvedimenti in modo da evitare conflitti per l'uso del bus.
- Questo problema di coordinare l'attività dei dispositivi Master in modo che usino il bus uno alla volta senza conflitti per l'uso dei fili condivisi viene detto **arbitrazione del bus**. Ovviamente i dispositivi che risolvono il problema vengono detti **arbitri**.

Arbitrazione di più Masters

- Il meccanismo normalmente adottato per realizzare un arbitro é quello della preassegnazione di **priorità** diverse ai diversi Master. Le richieste vengono esaminate ed il bus viene assegnato al Master con priorità più alta tra quelli che lo hanno richiesto.
- Una volta ottenuto il bus, un Master ne mantiene il controllo fino alla fine dell'operazione di lettura/scrittura, anche se nel frattempo arrivano altre richieste a priorità maggiore (servizio non interrompibile).

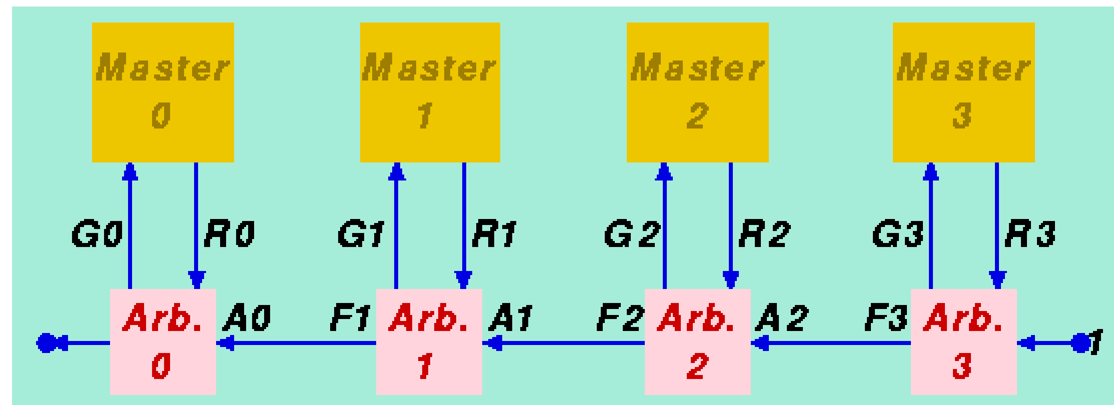
Arbitrazione di più Masters

- Il criterio di assegnazione di un valore di priorità più o meno alto ad un dispositivo ai fini dell'arbitrazione del bus non é legato alla "importanza" del dispositivo nel sistema, bensì alle sue esigenze in termini di tempo di risposta: vengono assegnate priorità alte a dispositivi che necessitano di un accesso immediato al bus e priorità più basse ai dispositivi che possono "tollerare" ritardi per l'acquisizione del bus senza comprometterne il funzionamento.

Arbitrazione di più Masters

- Teoricamente si potrebbe pensare di realizzare un arbitro di un bus come un unico dispositivo con tanti ingressi quanti sono i dispositivi Master (per riceverne le richieste) ed altrettante uscite (per inviare la conferma al Master che può iniziare ad usare il bus). Per motivi di semplicità realizzative e di economia di numero di fili, tuttavia, di solito si preferisce realizzare una versione **distribuita** di un arbitro a priorità, detta realizzazione "daisy-chain".

Arbitrazione di più Masters



- Ciascun Master (i) é collegato ad un proprio Arbitro (i) mediante un filo di richiesta Ri (1=richiesta, 0=non serve il bus) ed un filo di concessione del bus Gi (1=puoi diventare Master del bus, 0=devi aspettare). L'i-esimo arbitro riceve una abilitazione Ai dall'arbitro a priorità immediatamente superiore (i+1) (Ai=1 significa "se il tuo Master chiede il bus glielo puoi concedere subito"; Ai=0 significa "se il tuo Master chiede il bus devi farlo aspettare"). Inoltre, l'i-esimo arbitro manda in uscita all'arbitro a priorità immediatamente inferiore una segnalazione circa la disponibilità del bus Fi (1=il bus può essere assegnato a Master di priorità inferiore, 0=il bus é già stato assegnato ad un Master di priorità maggiore o uguale ad i).

Trasferimenti DMA

- L'uso di molti dispositivi all'interno di un sistema di calcolo prevede il trasferimento di quantità notevoli di informazioni da/verso la memoria RAM. Esempi tipici sono i dispositivi di comunicazione in rete che prevedono la trasmissione e la ricezione di messaggi e le unità a disco (magnetico, ottico, ecc.). Tipicamente l'uso efficiente di questi dispositivi prevede il trasferimento di "blocchi" di dati codificati in molte celle di memoria di indirizzo consecutivo.

Trasferimenti DMA

- Ovviamente il trasferimento di dati tra il "drive" di un dispositivo e la memoria potrebbe essere effettuato dalla CPU mediante l'esecuzione di un apposito programma che legge un dato alla volta dalla RAM o da un registro del drive, lo copia in un suo registro interno, lo riscrive in un registro del drive o in una cella di RAM, e ripete la sequenza di lettura e riscrittura fin quando l'intero blocco di dati non é stato trasferito.

Trasferimenti DMA

- Tale realizzazione (normalmente detta mediante "programmed I/O") risulta però gravemente inefficiente. Una alternativa decisamente superiore dal punto di vista della velocità di completamento del trasferimento e di uso equilibrato dei dispositivi contenuti nel sistema di calcolo é il cosiddetto **Direct Memory Access** (DMA) da parte del drive del dispositivo coinvolto nell'operazione.

Trasferimenti DMA

- L'idea di base di un trasferimento DMA é quella di dotare il drive di un dispositivo della capacità di indirizzare la memoria RAM per realizzare delle sequenze di cicli di lettura o di scrittura. Quando l'esecuzione di un programma da parte della CPU richiede un trasferimento di dati tra la RAM ed il dispositivo, la CPU si limita a "programmare" il trasferimento mandando un opportuno comando al drive nel quale viene specificato l'indirizzo della prima cella di memoria coinvolta, il numero totale di celle da trasferire, e la direzione del trasferimento (dal dispositivo verso la RAM o viceversa).

Trasferimenti DMA

- Fatto questo, la CPU può procedere oltre nell'esecuzione del programma (ovviamente senza poter far affidamento sul fatto che il trasferimento sia stato completato), e sarà il drive del dispositivo a portare a termine effettivamente il trasferimento in modalità DMA. Per fare questo, il drive dovrà poter diventare Master del bus in alternativa alla CPU. Quindi la realizzazione di un sistema che ammette l'uso di trasferimenti DMA richiede un opportuno protocollo di arbitrazione per il bus, come abbiamo visto in precedenza.

Trasferimenti DMA

- Nei sistemi di calcolo reali, il concetto di trasferimento DMA viene di solito realizzato in due modi diversi: mediante l'uso di **canali DMA** oppure di tecniche di **Bus Mastering**. La seconda alternativa corrisponde ad una realizzazione piena dell'idea di trasferimento DMA così come illustrata sopra in termini intuitivi, mentre la prima ne costituisce una approssimazione, caratterizzata da alcune limitazioni di funzionalità che consentono un risparmio nella complessità di realizzazione.

Trasferimenti DMA canali DMA

- Un canale DMA sostituisce la CPU nel trasferimento dei dati tra memoria e drive del dispositivo. Normalmente a livello di sistema vengono definiti uno o più canali DMA, che sono dei semplici "co-processori" specializzati nel trasferimento di dati, ma non legati a nessun particolare drive di dispositivi. La CPU programma un canale DMA specificando, oltre ai parametri relativi agli indirizzi di RAM, anche l'identificazione del registro del drive nel/dal quale scrivere/leggere i dati.

Trasferimenti DMA
canali DMA

- Il drive stesso si comporta da Slave negli accessi ai suoi registri operati dal canale DMA. É il canale DMA e non il drive del dispositivo che "compete con la CPU" per diventare Master del bus. Con uno schema realizzativo di questo tipo, l'aggiunta di nuovi drive connessi al bus non richiede complicazioni nell'arbitrazione del bus.

Trasferimenti DMA
Bus Mastering

- Un drive con capacità di Bus Mastering può invece essere programmato dalla CPU per diventare lui stesso Master del bus ed effettuare i trasferimenti DMA, senza dover far affidamento su nessun "canale" esterno. Quindi ogni drive deve contenere al suo interno la logica di indirizzamento della memoria e di interfacciamento con l'arbitro del bus.

Trasferimenti DMA
Bus Mastering

- L'aggiunta di un nuovo drive nel sistema aumenta il numero di potenziali Master del bus, e può quindi essere realizzata in modo semplice e modulare solo utilizzando una tecnica di arbitrazione distribuita tipo daisy-chain.

Memoria Virtuale

- Un'altra funzionalità ritenuta ormai fondamentale per la realizzazione di un sistema di calcolo moderno é la cosiddetta memoria virtuale. Con questo termine si intende la possibilità di gestire la memoria in modo molto più strutturato ed ordinato rispetto all'idea originale di Von Neumann. Se da un lato la disponibilità di un'unica RAM nella quale si possono mischiare a piacere dati e codice é concettualmente semplice e teoricamente molto potente, in pratica un uso disordinato della memoria da parte dei programmi é fonte inesauribile di errori di programmazione oltre che un grave impedimento per la possibilità di ottimizzare l'uso della memoria in modo automatico. Molto meglio sarebbe se ciascun programma potesse usufruire di un insieme di RAM diverse per soddisfare le diverse esigenze di memorizzazione di codice e dati.

Memoria Virtuale

- La memoria virtuale realizza proprio questa funzionalità: fa "vedere" al programma in esecuzione un insieme di moduli di memoria distinti e con diverse caratteristiche di accesso ai dati, supportate da dispositivi fisici specializzati, mentre continua ad utilizzare i soliti moduli RAM a livello fisico per questioni di economia di realizzazione.

Memoria Virtuale

- Un altro aspetto che ha portato storicamente allo sviluppo della memoria virtuale é la possibilità di sfruttare una unità a disco detta **swap** per simulare la disponibilità di dispositivi RAM di dimensioni superiori a quelli effettivamente disponibili nel sistema.

Segmentazione

- La segmentazione (pura) é considerata la tecnica più semplice per la realizzazione di un meccanismo di memoria virtuale. A livello di macchina convenzionale viene definito un insieme di **segmenti**, che possono essere immaginati come moduli di memoria distinti, ciascuno in grado di supportare modi di accesso specializzati e ciascuno caratterizzato dal suo proprio "spazio di indirizzamento" indipendente.

Segmentazione

- Chiariamo il concetto mediante un esempio molto semplice. Consideriamo il caso di una macchina convenzionale che fa riferimento a tre segmenti diversi, che chiameremo "segmento codice", "segmento dati statici" e "segmento stack".

Segmentazione

- Il segmento codice sarà ovviamente destinato a contenere la codifica binaria delle istruzioni del programma da eseguire, e nessun altro tipo di informazioni. Al segmento codice la CPU potrà sensatamente fare accesso solo durante la fase di fetch delle istruzioni, e solo mediante operazioni di lettura.

Segmentazione

- Il segmento stack sarà destinato a contenere le celle di memoria da usarsi per la realizzazione della struttura stack, per la memorizzazione di dati locali di una procedura, il passaggio di parametri, il salvataggio di valori contenuti in registri della CPU, ecc. La CPU potrà sensatamente accedere a tale segmento solo durante la fase di esecuzione di alcune istruzioni (JSR, RTS, ecc.), con modalità di lettura o scrittura.

Segmentazione

- Il segmento dei dati statici sarà destinato a contenere tutti gli altri dati del programma non memorizzati nello stack, e la CPU potrà sensatamente accedere a questo segmento solo durante la fase di esecuzione di altre istruzioni (per esempio le istruzioni MOVE).

Segmentazione

- Il fatto che i segmenti siano "noti a livello di macchina convenzionale" rende possibile un controllo da parte dei dispositivi che i vincoli di uso dei diversi segmenti siano rispettati dal programma in fase di esecuzione: la violazione di qualcuna delle restrizioni prestabilite per l'accesso ai dati contenuti nei vari segmenti (per esempio il fatto che si acceda in scrittura durante la fase di esecuzione di una istruzione MOVE ad una cella appartenente al segmento codice) può essere riconosciuta da dispositivi a livello di microarchitettura e "segnalata" sotto forma di trap, consentendo quindi alla CPU di interrompere l'esecuzione del programma che violerebbe le regole e di mandare invece in esecuzione un'apposita procedura di gestione definita dal sistema operativo.

Segmentazione

- L'idea astratta di "segmentazione" può essere concretamente definita in due modi diversi:
- Segmentazione implicita
  - Come nell'esempio appena abbozzato, i segmenti e la loro utilizzazione sono definiti una volta per sempre a livello di macchina. Questo fa sì che non ci sia bisogno, nella maggior parte dei casi, di dire esplicitamente a quale segmento ci si riferisce: é la natura stessa dell'operazione che stiamo eseguendo che implicitamente definisce anche il segmento su cui dobbiamo operare.

Segmentazione

- Segmentazione esplicita
  - In questo caso a livello di macchina specifichiamo solo che possiamo usare un insieme di segmenti dicendo qual é il massimo numero di segmenti che vogliamo usare. A priori però non specifichiamo a quale scopo siano destinati i segmenti. Sarà cura del programmatore stabilire quanti e quali segmenti usare effettivamente e quali modalità di accesso consentire o vietare per i diversi segmenti.

Segmentazione

- In questo caso il segmento su cui operare deve essere definito esplicitamente come parte dell'indirizzo delle celle di memoria. Tipicamente l'adozione di un meccanismo di segmentazione esplicita comporta la suddivisione della rappresentazione binaria dell'indirizzo di una cella di memoria in due sottoinsiemi di bit: un sottoinsieme rappresenta il numero del segmento a cui si vuole accedere, mentre l'altro sottoinsieme rappresenta il vero e proprio indirizzo della cella di memoria all'interno di quel segmento.

Segmentazione

- Come si diceva in precedenza, la struttura dei moduli RAM a livello fisico non cambia, quindi per realizzare un meccanismo di segmentazione (implicita o esplicita) occorre aggiungere dei dispositivi in grado di tradurre **indirizzi virtuali** (detti anche **indirizzi logici**) usati dal programma in esecuzione sulla CPU in **indirizzi fisici** usati sul bus per accedere alla RAM.

Segmentazione

- Nel caso della segmentazione implicita l'unico vero problema da risolvere per definire uno schema di traduzione da indirizzi logici in indirizzi fisici consiste nel trovare un modo semplice per tradurre un qualunque indirizzo compreso tra 0 ed N-1 (dove N indica la dimensione del segmento che stiamo considerando) in un indirizzo fisico di RAM. La soluzione normalmente adottata fa uso di una coppia di registri associata a ciascun segmento. Tali registri sono detti **registro base** e **registro limite**.

Segmentazione

- Il registro base contiene l'indirizzo della cella di RAM corrispondente alla cella di indirizzo logico 0 del segmento. Il registro limite contiene l'indirizzo della cella di RAM corrispondente alla cella di indirizzo logico N-1 del segmento (ossia l'ultima). La traduzione da indirizzo logico in indirizzo fisico consta di due passi: prima si somma all'indirizzo logico il contenuto del registro base, ottenendo l'indirizzo fisico corrispondente; poi si confronta l'indirizzo fisico col contenuto del registro limite, e se l'indirizzo fisico é maggiore del limite allora si genera una condizione di errore (si segnala una Trap).

Segmentazione

- Un problema rilevante che si pone per una gestione efficace di una memoria virtuale a segmentazione (pura) é quello di sfruttare bene la capacità della memoria fisica a disposizione anche in presenza di una variazione dinamica delle esigenze. Un esempio del problema si trova pensando al segmento "stack", la cui dimensione ottimale non é semplice da stabilire a priori e una volta per tutte.

Segmentazione

- La quantità di celle di memoria utilizzate nella struttura stack varia dinamicamente durante l'esecuzione del programma, ed il segmento dovrebbe essere definito di ampiezza tale da contenere tutte le celle nel momento della massima estensione dello stack.

Segmentazione

- Abbondare troppo nella dimensione rispetto a questo requisito porterebbe ad uno spreco di memoria che poi non verrebbe effettivamente usata dal programma, mentre una approssimazione per difetto della dimensione ottimale provocherebbe ad un certo punto la condizione di errore per superamento delle dimensioni del segmento.

Segmentazione

- Pragmaticamente la soluzione adottata potrebbe consistere nel definire un segmento stack di dimensioni "medie" (in modo da non sprecare troppa RAM), con la possibilità però di estendere dinamicamente la dimensione del segmento in caso di superamento da parte dello stack delle dimensioni prestabilite (cosa che può essere gestita mediante il meccanismo delle Trap).

Segmentazione

- L'estensione di un segmento richiede però la disponibilità di celle di memoria nella RAM con indirizzi **contigui** rispetto a quelli delle celle facenti già parte dello stesso segmento. Quindi potrebbe succedere che nel momento in cui si vuole estendere un segmento le celle di indirizzo adiacente siano "usate" per realizzare altri segmenti, mentre altre celle di RAM con indirizzi non adiacenti siano libere. In questi casi occorrerebbe "spostare" i segmenti (copiando da una cella all'altra i dati già memorizzati nei segmenti usati) per far in modo che le celle di indirizzo contiguo vengano "liberate" e rese quindi disponibili per l'estensione.

Segmentazione

- Queste operazioni di copia di dati provocherebbero ovviamente rallentamenti nella gestione dell'eccezione di ampliamento del segmento; per questo nei sistemi moderni si cerca di evitarle introducendo il meccanismo della paginazione

Paginazione

- Introducendo il meccanismo della segmentazione ci siamo scontrati col problema di gestione efficiente della memoria dovuto al vincolo di contiguità degli indirizzi delle celle di memoria fisica da utilizzare per realizzare un segmento. Tale vincolo deriva dalla semplicità dell'algoritmo di traduzione da indirizzo logico ad indirizzo fisico (la somma di una costante, detta "indirizzo base"). Ovviamente si può pensare che, complicando un po' l'algoritmo di traduzione sia possibile superare questo vincolo di contiguità sugli indirizzi fisici (pur mantenendo la contiguità degli indirizzi logici).

Paginazione

- In effetti una soluzione concettualmente semplice per eliminare totalmente tale vincolo sarebbe quella di definire una funzione uno-uno tra gli indirizzi logici consecutivi del segmento (0, 1, 2, ... N-1) e qualsiasi sequenza di N indirizzi fisici della RAM. In tal modo si potrebbe per esempio specificare la corrispondenza tra l'indirizzo logico 0 e l'indirizzo fisico 31415, tra l'indirizzo logico 1 e l'indirizzo fisico 2047, tra l'indirizzo logico 2 e l'indirizzo fisico 3, ecc.

Paginazione

- La definizione di tale funzione sarebbe supportata da un vettore di N elementi, che riporti gli indirizzi fisici corrispondenti agli indirizzi logici i cui numeri corrispondono alle posizioni (nel nostro esempio, il vettore [31415, 2047, 3, ...]). Volendo tradurre l'indirizzo logico "1" in indirizzo fisico non dovremmo far altro che estrarre il valore dell'elemento numero "1" dal vettore che rappresenta la nostra funzione, per capire che l'indirizzo fisico corrispondente é "2047".

Paginazione

- Ovviamente la soluzione che stiamo prospettando è valida solo da un punto di vista teorico, in quanto la sua realizzazione pratica sarebbe altamente inefficiente. La tabella di traduzione degli indirizzi dovrebbe essere memorizzata da qualche parte (tipicamente un dispositivo RAM, e per indirizzare in modo virtuale una RAM da N celle ci servirebbe un'altra RAM da N celle che contenga la tabella: quindi potremmo usare per scopi utili solo metà delle celle di RAM effettivamente presenti nel sistema. Ci serve quindi una soluzione concettualmente simile ma che richieda la memorizzazione di un vettore di indirizzi fisici di dimensione nettamente inferiore rispetto al numero di celle di RAM che vogliamo indirizzare in modo virtuale.

Paginazione

- La soluzione praticamente accettabile viene ottenuta ponendo dei vincoli sui valori che la funzione uno uno può associare ai vari indirizzi logici. Per esempio, potremmo stabilire che l'indirizzo logico 0 possa corrispondere ad un qualsiasi indirizzo fisico J, ma che poi l'indirizzo logico 1 non possa corrispondere che all'indirizzo fisico J+1.

Paginazione

- Analogamente, potremo scegliere in modo completamente arbitrario l'indirizzo fisico K da associare all'indirizzo logico 2, ma con questo avremo anche stabilito una volta per tutte che l'indirizzo logico 3 corrisponde all'indirizzo fisico K+1.
- Detto in altri termini, potremmo specificare una funzione uno-uno arbitraria che associa a ciascun valore pari di indirizzo logico un qualsiasi valore di indirizzo fisico, ed useremo questa stessa funzione per calcolare la corrispondenza anche per gli indirizzi logici dispari (oltre che per quelli pari), semplicemente sommando "1" al valore della funzione applicata all'indirizzo logico di valore pari immediatamente precedente.

Paginazione

- La traduzione richiederà una operazione di somma in più nel caso dispari rispetto alla funzione completa, per tutti gli indirizzi, ma il vettore richiederà un numero di elementi che é solo la metà del numero di celle di RAM da usare. Quindi, avendo a disposizione un numero totale di M celle di RAM per la realizzazione di una memoria virtuale, potremmo usarne N=2/3 M come celle effettivamente utili, e solo M/3 per la memorizzazione del vettore di traduzione.

Paginazione

- Il prezzo che dobbiamo pagare per questa maggior efficienza nell'uso della memoria fisica consiste nella necessità di dover sempre considerare due celle di memoria fisica adiacenti. Se anche a noi servisse una sola cella di memoria per memorizzare i nostri dati, saremmo comunque costretti ad "impegnare" anche la traduzione dell'indirizzo della cella di indirizzo fisico successivo, che quindi non sarebbe più utilizzabile per la realizzazione fisica di un indirizzo virtuale diverso dal successore dell'indirizzo logico corrispondente alla cella di indirizzo fisico precedente.

Gerarchia di Memoria e Memorie Cache

- Abbiamo visto che l'introduzione di un meccanismo di memoria virtuale, indispensabile per offrire protezione e utile per semplificare il problema dell'allocazione di memoria, può determinare un rallentamento degli accessi alla memoria. Per ovviare a questo inconveniente, e più in generale per cercare di aumentare la velocità di esecuzione dei programmi, vediamo ora l'idea di uso delle memorie Cache.

Gerarchia di Memoria e Memorie Cache

- Chiunque abbia provato a organizzare un programma da eseguire su una macchina convenzionale prima o poi potrebbe cominciare a chiedersi se l'uso di una memoria RAM omogenea sia proprio così sensato ed efficace come può sembrare di primo acchito. Indubbiamente se si pensa ad un qualunque programma particolare é ben difficile individuare delle modalità di uso omogenee per le celle di memoria di indirizzo diverso. Inevitabilmente alcune celle di memoria verranno usate per memorizzare dati o istruzioni alle quali sarà necessario accedere più spesso che non ad altre.

Gerarchia di Memoria e Memorie Cache

- Pertanto, in presenza di diverse tecnologie realizzative caratterizzate da diversi costi e diversi tempi di accesso ai dati, potrebbe apparire più conveniente adattare in qualche modo l'allocazione dei dati su un insieme di moduli di memoria eterogenei.

Gerarchia di Memoria e Memorie Cache

- Grosse quantità di dati a cui si accede con scarsa frequenza potrebbero essere memorizzati in moduli RAM dinamici caratterizzati da basso costo e tempi di accesso relativamente lunghi, mentre piccole quantità di dati a cui si accede con maggior frequenza potrebbero essere più convenientemente memorizzati in moduli RAM statici caratterizzati da tempi di accesso molto inferiori (anche se da costo più elevato per bit).

Gerarchia di Memoria e Memorie Cache

- Scopo dell'introduzione di una **memoria cache** é proprio quello di sfruttare le diverse caratteristiche tecnologiche e di costo dei dispositivi di memoria, abbinandole nel modo economicamente più conveniente in modo da ottenere in media il minor tempo di accesso possibile ai dati. Il nome deriva da un termine francese (per una volta non dall'inglese ...) che significa "nascosta". Infatti non si vuole che questa ottimizzazione implichi un maggior onere per il programmatore, ma si pretende addirittura che sia il sistema stesso ad "indovinare" quali dati é meglio inserire nella piccola memoria veloce e quali invece possono essere lasciati nella grande memoria meno veloce senza grosso impatto sulle prestazioni.

Gerarchia di Memoria e Memorie Cache

- I programmi vengono quindi organizzati sempre nel solito modo, pensando di accedere ad una unica memoria RAM omogenea. Durante l'esecuzione del programma il sistema di controllo della cache "decide autonomamente" di copiare parte dei dati usati più frequentemente dal programma nella memoria cache per ridurne il tempo di accesso.

Gerarchia di Memoria e Memorie Cache

- Questo implica che il cache controller sia interposto, a livello di microarchitettura, tra la CPU e la memoria RAM. Quindi la CPU "crede di indirizzare la RAM", mentre in realtà comunica al cache controller le proprie richieste di accesso alla memoria. Il cache controller, sulla base della storia di tutte le richieste di accesso derivanti dalla CPU decide quali dati vale la pena copiare in cache e quali no.

Gerarchia di Memoria e Memorie Cache

- Il cache controller risponde alle richieste della CPU o immediatamente nel caso si richieda l'accesso ad un dato copiato in memoria cache, oppure con ritardo nel caso si richieda l'accesso ad un dato contenuto solo in RAM. In quest'ultimo caso il dato viene reperito dal cache controller indirizzando la cella opportuna di memoria.
- Dal punto di vista della memoria RAM, é quindi il cache controller che prende il posto della CPU come dispositivo Master sul Bus per comandare le operazioni di lettura o scrittura.

Gerarchia di Memoria e Memorie Cache

- L'individuazione di un algoritmo efficiente eseguibile dal controllore di cache per "decidere" se copiare un dato in cache oppure no é un problema di non banale soluzione.

Gerarchia di Memoria e Memorie Cache

- Il primo passo verso la soluzione del problema consiste nello studio delle caratteristiche generali dei programmi che devono essere eseguiti dalla CPU per cercare di individuarne proprietà comuni e regolarità di comportamento su cui poter fare affidamento.

Gerarchia di Memoria e Memorie Cache

- Effettuando studi di tipo statistico su un gran numero di programmi anche molto diversi tra loro si arrivò a determinare una coppia di caratteristiche generali che tutti i programmi esibiscono in modo più o meno accentuato. Tali caratteristiche sono note come proprietà di **località** dei programmi.

Gerarchia di Memoria e Memorie Cache

- **Località nel tempo**:
  - se un programma ad un certo punto richiede l'accesso ad una particolare cella di memoria di indirizzo I, allora é molto probabile che dopo un breve intervallo di tempo lo stesso programma richieda nuovamente l'accesso alla stessa cella di memoria di indirizzo I.
- **Località nello spazio**:
  - se un programma ad un certo punto richiede l'accesso ad una particolare cella di memoria di indirizzo I, allora é molto probabile che dopo un breve intervallo di tempo lo stesso programma richieda l'accesso anche ad un'altra cella di memoria di indirizzo "adiacente" J=I+d (ovvero con "d" numero positivo o negativo molto piccolo).

Gerarchia di Memoria e Memorie Cache

- L'esistenza di queste proprietà di località dei programmi rende effettivamente perseguibile l'idea di un algoritmo di scelta del sottoinsieme di dati da copiare in Cache basato sull'analisi dei precedenti accessi alla memoria da parte dello stesso programma.

Gerarchia di Memoria e Memorie Cache

- In mancanza di proprietà di questo tipo esibite dai programmi avrebbe poco senso tener conto di quali dati ha usato il programma fino ad un certo punto per cercare di indovinare quali userà da quel punto in avanti: tanto varrebbe tirare ad indovinare senza tener conto di quanto é successo prima!

Gerarchia di Memoria e Memorie Cache

- La proprietà di località nel tempo ci suggerisce ovviamente di copiare in Cache il contenuto di una cella di memoria nel momento in cui la CPU chiede di accedervi. Se il programma gode effettivamente della proprietà di località nel tempo rispetto a quella cella di memoria, allora nell'immediato futuro ripeterà l'accesso allo stesso dato, e la prossima volta la risposta da parte del controllore di cache potrà essere immediata, senza più richiedere un ulteriore accesso allo stesso dato in RAM.

Gerarchia di Memoria e Memorie Cache

- Quindi il controllore di Cache può essere realizzato in modo da partire dalla situazione di Cache vuota e, tutte le volte che la CPU chiede di accedere ad un "nuovo" dato (ossia ad un indirizzo corrispondente ad una cella di RAM il cui valore non sia già contenuto anche nella Cache), indirizzare la RAM, recuperare il dato e, oltre che fornirlo alla CPU, memorizzarlo anche nella Cache.

Gerarchia di Memoria e Memorie Cache

- Vista la limitata dimensione della memoria Cache rispetto alla RAM, anche partendo da una situazione di Cache vuota si arriverà prima o poi al punto in cui la Cache viene riempita completamente, e quindi per inserirvi un nuovo dato occorrerà "far spazio" cancellandone un altro inserito in precedenza.
- Si pone quindi il problema di trovare un buon algoritmo di **rimpiazzamento** per decidere quali tra i dati già contenuti in Cache deve essere eliminato per far posto ad un nuovo dato.
- Il problema é stato studiato estensivamente, e la miglior soluzione trovata é il cosiddetto **algoritmo LRU** (dall'inglese "Least Recently Used"), ossia la scelta dell'elemento usato meno recentemente.

Gerarchia di Memoria e Memorie Cache

- Fin qui abbiamo discusso la possibilità di trarre vantaggio dalla proprietà di località nel tempo mediante l'aggiunta in Cache di un nuovo valore tutte le volte che la CPU accede ad un dato non contenuto nella Cache stessa, con eventuale rimpiazzamento di un altro valore scelto mediante l'algoritmo LRU (... o meglio, in pratica, di sue approssimazioni più facili e meno costose da realizzare). Tuttavia é semplice sfruttare anche le caratteristiche di località nello spazio.
- L'algoritmo che sfrutta le proprietà di località nello spazio consiste nell'anticipare le future richieste della CPU copiando in Cache non solo il dato che ha richiesto, ma anche quelli contenuti in celle di RAM di indirizzo "abbastanza vicino" (senza ovviamente esagerare ...).

Gerarchia di Memoria e Memorie Cache

- La soluzione (molto semplice) adottata in pratica per trarre vantaggio dalle caratteristiche di località nello spazio consiste nel definire delle **linee di cache** che raggruppano più di una parola di memoria di indirizzo consecutivo. Interagendo con la RAM, il controllore di Cache legge e scrive blocchi di dati pari ad una intera linea di cache, non singole celle di memoria. Quindi quando copia il valore contenuto in una cella di memoria, in realtà copia anche i valori delle altre celle di memoria di indirizzo contiguo che costituiscono la stessa linea.

Gerarchia di Memoria e Memorie Cache

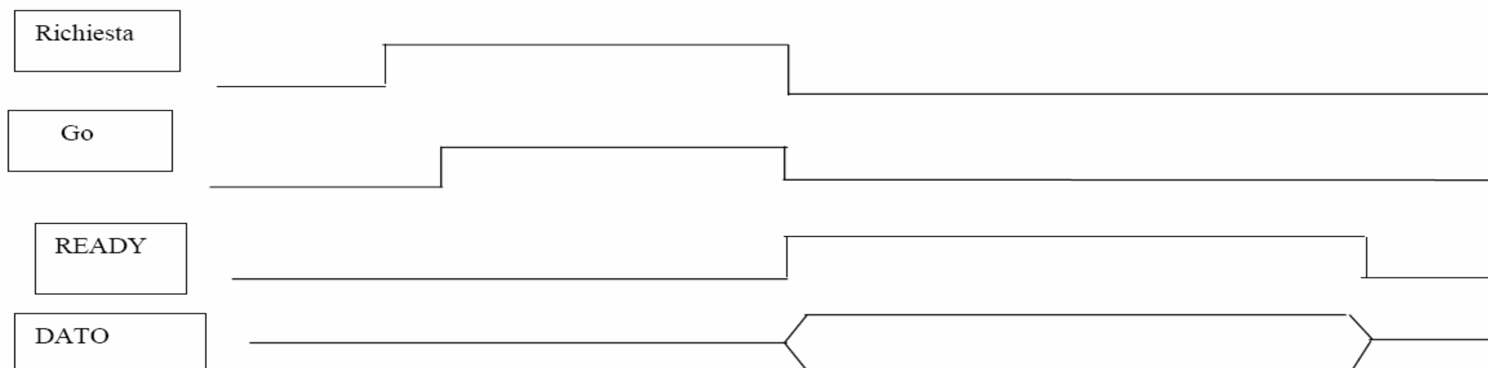
- Per i soliti ovvi motivi di semplicità realizzativa del trattamento degli indirizzi espressi in forma binaria, la dimensione di una linea di cache viene sempre scelta tra le potenze di 2. In analogia col meccanismo della memoria virtuale a paginazione, questo permette di suddividere la rappresentazione binaria di un indirizzo in due campi, che possono essere interpretati come un numero di linea ed uno spiazzamento all'interno della linea stessa.

Gerarchia di Memoria e Memorie Cache

- Una delle differenze praticamente più rilevanti tra linee di cache e pagine in un sistema di memoria virtuale a paginazione é costituita dalle dimensioni: mentre per le pagine valori normalmente utilizzati variano tra 256 e 4096 parole, per le linee di cache le dimensioni variano solitamente tra 2 e 8 parole.

Esercizio 20.1

- Scrivere un sottoprogramma che acquisisca da una periferica di input una sequenza di n lettere maiuscole e li memorizzi in memoria a partire da un indirizzo assegnato.
- Il sottoprogramma accetta come parametri di ingresso il numero di caratteri (n), l'indirizzo iniziale dell'area di memoria. Il driver termina la lettura solo quando sono stati acquisiti n lettere maiuscole. Il driver restituisce al programma chiamante il numero di caratteri scartati (quelli diversi dalle lettere maiuscole). Il protocollo utilizzato per la lettura del singolo carattere è quello specificato dal diagramma riportato in figura. Il registro di controllo e stato è mappato in memoria all'indirizzo \$8100 e il registro dato all'indirizzo \$8101. Il bit di richiesta, il bit Go e il bit ready sono rispettivamente il bit 0, il bit 1 e il bit 7 del registro di controllo e stato.
- N.B. Il bit di richiesta e go si abbassano nel momento in cui diventa alto il bit ready e quindi non richiedono un comando esplicito del processore.



Codice ASCII

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.pubblinet.com