UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Ingegneria Informatica

Orale Calcolatori Elettronici

Domande e Risposte

Walter Soro

Anno Accademico: 2022/2023

Indice

INTRODUZIONE

Ho scritto questi appunti mentre mi preparavo per l'orale quindi prendete le risposte con le pinze. L'idea in generale è stata quella di completare tutti gli argomenti sotto forma di domanda in modo da farsi uno schema mentale delle cose da dire per argomento ed eventuali domande che potrebbe fare. L'ultima sezione invece sono domande prese da un orale precedente. Buona Fortuna!

1 CACHE

1.1 Come mai è stata introdotta la CACHE?

Le scritture in Memoria RAM possono richiedere oltre 200 cicli di clock, e un programma ci accede molte volte. Quindi è stata aggiunta una nuova memoria molto più piccola e costosa ma più veloce, la memoria cache. L'idea è quella di salvare in memoria il contenuto di indirizzi utilizzati di recente e che si pensa il programma riusi nelle prossime istruzioni, in questo modo non c'è bisogno di accedere sempre alla memoria lenta ma solo alla cache. La soluzione utilizzata è quindi di usare entrambe le memorie.

1.2 A cosa serve la CACHE?

Serve a migliorare le prestazioni in quanto è una memoria molto più veloce della RAM.

1.3 Quali sono i principi su cui si basa il salvataggio in CACHE?

I principi utilizzati sono quelli di:

- **Località Spaziale**: Se la CPU accede ad un indirizzo è probabile che successivamente acceda ad indirizzi vicini
- **Località Temporale**: Se la CPU accede ad un indirizzo è probabile che a breve ci acceda di nuovo

Sono principi con i quali si cerca di predire il comportamento successivo della CPU e si basano su alcune considerazioni:

- I programmi sono spesso eseguiti in sequenza quindi è probabile che dopo una istruzione la prossima sia all'indirizzo successivo
- Nei programmi con dei cicli si vorrà leggere più volte allo stesso indirizzo
- Nei programmi spessi si usano strutture dati le cui componenti sono salvate in indirizzi vicini

1.4 Chi si occupa di salvare in cache?

Il Controllore Cache.

1.5 Cosa è il controllore cache?

Il controllore cache è un circuito montato a valle del processore e a monte del controllore bus (nella architettura senza MMU).

1.6 Cosa fa il controllore cache?

Il controllore cache si occupa della gestione della cache:

- Intercetta tutte le operazioni di lettura e scrittura comandate dalla CPU
- Esamina l'indirizzo di ogni operazione e controlla se si trova in memoria cache, se è così allora completa l'operazione con il dato salvato in cache, altrimenti esegue l'operazione in RAM al posto della CPU e salva il contenuto in cache. Per il principio di località spaziale preleva più locazioni di quelle che servono, l'unità spaziale è la cacheline in genere 64byte. Quando viene richiesta una particolare locazione viene trasferita tutta la cacheline che la contiene
- Quando la memoria cache è piena decide quali locazioni mantenere e quali eliminare.
- Fa passare tutte le operazioni di I/O senza intercettarle

1.7 Come funziona il meccanismo della cache?(Non sapevo come scriverla)

Il meccanismo della cache si svolge tutto in hardware tramite il controllore cache, inoltre questo meccanismo è trasparente al processore, infatti non serve modificare la CPU per utilizzare la cache tranne un piedino che segnale quando il controllore ha terminato l'operazione o se deve essere prolungata. Il meccanismo cache è completamente trasparente anche al Software, infatti i programmi ignorano completamente la presenza della cache.

1.8 Come funziona la CACHE?

Quando la CPU genera una operazione in memoria, pone sulle sue linee A23_A0 l'indirizzo di una parola quadrupla allineato naturalmente e sui i bit /B7_B0 i suoi bit enable per scegliere i byte coinvolti nella riga.
Il controllore cache per le cacheline a 64byte divide l'indirizzo in 2 parti:

1. OFFSET 3 bit meno significativi
2. Indirizzo di linea 53-3 bit più significativi (indirizzi fisici)

Il controllore cache usa l'indirizzo di linea per capire se la locazione di memoria è presente o meno in cache, e se è presente a che indirizzo si trova.
Un modo per fare questa operazione è sufficiente avere una funzione hash da indirizzi di riga ad indirizzi di cache.

1.9 Come funziona la cache ad indirizzamento Diretto?

1.9.1 Come viene diviso l'indirizzo?

Il controllore cache intercetta i 61 bit di indirizzo che genera la CPU. Sia 2^a il numero di cacheline, il controllore divide l'indirizzo in questo modo

- 3 bit meno significati per l'**offset**

- a bit di **indice** per identificare la cacheline all'interno di una regione di 2^a cacheline
- 52-a bit per l'**etichetta** per identificare una regione di grande 2^a cacheline

1.9.2 Qual e il problema delle cache ad indirizzamento diretto?

Il problema della cache ad indirizzamento diretto è che gli indirizzi in memoria che distano 2^a cacheline hanno lo stesso indice (conflitto fra linee). L'etichetta serve a questo, infatti due cacheline che hanno lo stesso indice hanno però etichetta diversa.

1.9.3 Come è fatta?

Alla memoria cache è affiancata un'altra RAM detta memoria delle etichette che contiene per ogni indice:

- etichetta associata a quell'indice
- bit V detto bit di validità che indica se la cacheline è valida oppure no. All'inizio tutti i bit V sono a 0 e il controllore li setta quando carica le cacheline. Utile quando è necessario invalidare i dati in cache.
- In caso di politica Write back un bit D(dirty) che indica se la cacheline è stata modificata ma il suo contenuto non è stato caricato in memoria

L'etichetta in arrivo viene confrontata con l'etichetta in cache se sono uguali e la cacheline è valida, allora si ha una **hit**, altrimenti una **miss**.

Quindi con un circuito di uguaglianza controllo se le etichette sono uguali e poi il risultato lo metto in AND con il bit di validità corrispondente.

1.9.4 Cosa fa il controllore per le operazioni di read?

- **read hit**: il controllore completa l'operazione di lettura senza accedere alla RAM
- **read miss**: il controllore accede in RAM, recupera la cacheline e la salva in cache e poi si comporta come nel caso *read hit*

1.9.5 Cosa fa il controllore per le operazioni di write?

Per le operazioni di write ci sono alcune politiche che si possono applicare:

- **write hit**, si possono applicare 2 politiche:
  - **Write back**, viene aggiornata solo cache, il dato viene poi ricopiato in memoria quando la cacheline viene rimpiazzata. Questa politica è vantaggiosa perché è probabile che una cacheline venga modificata più volte prima di essere rimpiazzata.
  - **Write though**, viene aggiornata sia la cache che la memoria centrale

- **write miss**, si possono applicare 2 politiche:
  - **Write no-allocate**, completa la scrittura in memoria senza caricare la cache-line in cache
  - **Write allocate**, carica la cacheline in cache e poi si comporta come una *write hit*, quindi può scegliere se write back o write though

  Nel caso della politica *write back* si aggiunge alla memoria delle etichette un bit D(dirty) che se =1 indica che la cacheline è stata modificata ma non è stata copiata in memoria centrale e quindi prima di essere rimpiazzata deve essere copiata in RAM.

1.9.6 Disegno Cache ad indirizzamento diretto

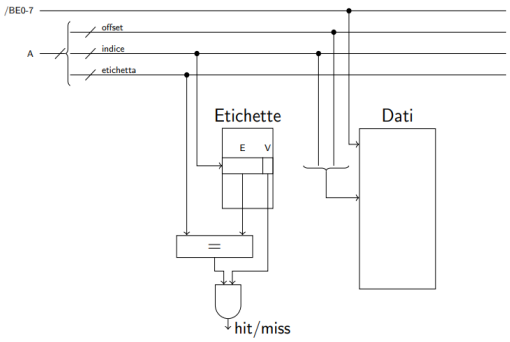


Figura 1: CACHE ad indirizzamento diretto

1.10 Come mai le cacheline sono a 64byte?

64byte sono un ottimo compromesso con il fatto che se facessi cacheline più grandi avrei una memoria delle etichette molto più piccola e il fatto che se sono troppo grandi leggere o scrivere una cacheline impiegherebbe troppo tempo.

1.11 Come funzionano le cache associative ad insiemi?

Le cache associative ad insiemi sono delle cache dirette montate in parallelo, quindi con n memorie delle etichette e n memorie dati. L'idea è quella di dividere le cacheline nelle varie tabelle dati. Ora un indice una possibile cacheline in una delle tabelle dati. Le uscite degli AND delle tabelle etichette vanno in tutte in un OR che rappresenta hit/miss e faranno anche da bit di comando di un multiplexer che prende tutte le uscite dei vari dati, infatti solo una delle uscite delle tabelle dati è valida. Questa architettura migliora il problema del conflitto fra linee fino ad eliminarlo completamente nel caso estremo di cache *completamente associativa* che è la cache con un solo insieme. Infatti aumentando il numero di vie diminuisce il numero di insiemi.
Il problema sta nel caso di *miss* quindi quando dovrò decidere quale etichetta rimpiazzare,

in questo caso si usa la polita LRU(Last Recently Used): si rimpiazza la cacheline della via che non è stata acceduta da più tempo. Per questa politica è stata aggiunta un'altra memoria R.

1.11.1 Come funziona la politica pseudo-LRU?

Prendiamo il caso di 4 vie: Ho 4 vie A B C D, le raggruppo a due a due A B e C D in memoria R conservo 3 bit b0 b1 b2.

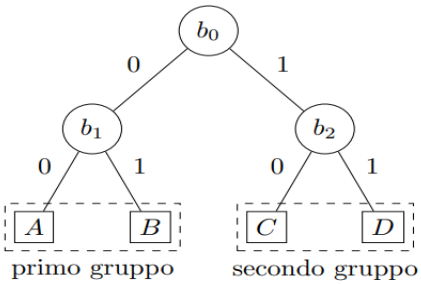


Figura 2: LRU a 4 vie

- Se b0 = 0 allora la via da rimpiazzare è una tra A e B altrimenti C e D.
- se b0 = 0 cerco tra A e B, controllo solo b1 ignorando b2, se b1 = 0 allora la via da rimpiazzare è A, altrimenti B.
- se b0 = 1 cerco tra C e D, controllo solo b2, se b2 = 0 allora la via è C altrimenti è D

1.11.2 Quale è il difetto della politica LRU?

1.12 Disegno CACHE associativa a 2 vie

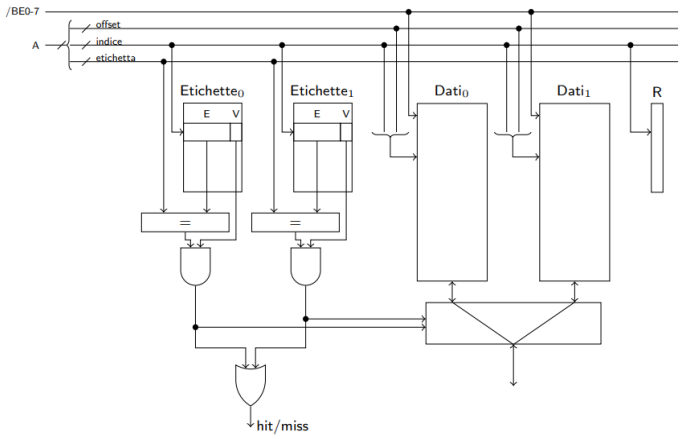


Figura 3: CACHE associativo ad insiemi a 2 vie

2 INTERRUZIONI

2.1 Perché sono state introdotte le interruzioni e a cosa servono?

Le interruzioni sono state introdotte per risolvere quelle problematiche di prestazioni che si hanno quando per esempio si usa una periferica. Usando una tastiera ad esempio senza le interruzioni dovremmo controllare con un ciclo infinito il registro RBR della tastiera fino a quando non ci si trova dentro un qualcosa di valido. Facendo così però la CPU non sta facendo altro che aspettare che questo accada bloccando tutte le altre funzionalità. Le interruzioni vengono introdotte per ovviare a questa problematica infatti ora quando la tastiera avrà un nuovo valore nel registro RBR si attiverà una interruzione che bloccherà il flusso normale del programma per leggere il tasto, così la CPU potrà lavorare in parallelo. Le interruzioni sono infatti un primo passo verso la multiprogrammazione.

2.2 A livello Hardware come ci si assicura che la CPU non veda 2 volte la stessa interruzione?

2.3 Nei processori INTEL come funziona il meccanismo per disattivare le interruzioni?

Nei processori INTEL la soluzione adottata è aggiungere alla CPU un flag IF (interrupt flag) in modo che la CPU ascolti le richieste di interruzione solo quando questo è settato. Al programmatore sono fornite le istruzioni cli e sti per resettare e settare il flag in questione. Le richieste in arrivo quando IF=0 verranno ascoltate quando IF tornerà ad 1.

2.4 Cosa è la IDT?

Ad ogni interruzione è associato un tipo che sta su 8 bit. I possibili tipi sono 2^8 (256) i primi 32 sono riservati alle eccezioni. La IDT (Interrupt Descriptor Table) è una tabella memorizzata in M1 contenente tutti i descrittori di interruzione quindi una entrata per ogni tipo possibile. La CPU accede alla IDT tramite il contenuto del registro /IDTR.

2.5 Cosa contiene un gate della IDT?

Ogni gate della IDT è formato da 16 byte 8 dei quali contengono l'indirizzo della routine associata all'interruzione. Contiene inoltre altre informazioni che servono a gestirne l'accesso.

- Bit P (present) se 0 significa che il gate non è presente
- Bit DPL (Descriptor Privilage Level) indica il livello minimo di priorità della CPU per poter attraversare il gate.
- Bit L indica il livello a cui la CPU dovrà andare dopo l'attraversamento

- Bit I/T che indica se il gate è di tipo Interrupt o Trap, se il gate è di tipo interrupt significa che dopo il passaggio bisogna disattivare le interruzioni resettando il BIT IF per bloccare le interruzioni mascherabili.

2.6 Cosa sono le interruzioni esterne?

Le interruzioni esterne sono le interruzioni che arrivano alla CPU tramite i piedini /INTR e /NMI e sono interruzioni che possono arrivare in qualsiasi momento durante l'esecuzione di una istruzione e per questo sono dette *asincrone*. Le interruzioni esterne possono essere di 2 tipi:

- **Non Mascherabili**, arrivano con il piedino /NMI, che significa che non possono essere ignorate, quindi il BIT IF per loro non conta. Hanno un tipo predefinito che è il 2.
- **Mascherabili**, arrivano tramite il piedino /INTR, e si traducono in una effettiva richiesta di interruzione solo se il flag IF in RFLAGS è 1. Il tipo dipende dall'interruzione ed è fornito dal controllore APIC.

Il processore prima di esaminare una interruzione esterna aspetta di concludere l'istruzione in corso. Quindi le interruzioni esterne **non possono bloccare l'esecuzione di una istruzione**.

2.7 Cosa sono le interruzioni software?

Sono prodotte dall'istruzione INT e vengono sollevate alla fine dell'esecuzione stessa, quindi sono *sincrone*, e **non possono sospendere l'esecuzione di una istruzione**. Le interruzioni software sono un modo per permettere all'utente di eseguire una routine di livello sistema in modo controllato.

2.8 Cosa sono le eccezioni?

Sono interruzioni prodotte dai circuiti di controllo interni della CPU e arrivano durante l'esecuzione dell'istruzione che l'ha causata, sono quindi *sincrone* rispetto al programma in esecuzione. Il tipo è implicito in base all'eccezione sollevate, ad esempio divisione per 0 tipo 0. Possono essere di 3 tipi:

- **Trap**, vengono sollevate tra l'esecuzione di una istruzione e la successiva. Quindi **non possono bloccare l'esecuzione di una istruzione**;
- **Fault**, vengono sollevate durante l'esecuzione di una istruzione in caso di errori recuperabili e per questo **sospendono l'esecuzione di una istruzione**.
- **Abort**, vengono sollevate durante l'esecuzione di una istruzione in caso di errori gravi e irrecuperabili, causandone la terminazione. Per questo **sospendono l'esecuzione di una istruzione**.

2.9 Che cosa fa in sequenza la CPU quando arriva una interruzione?

1. Si procura il tipo dell' interruzione e lo usa per accedere alla tabella IDT
   - Se sono interruzioni esterne riceve il tipo dall'APIC
   - Se sono interruzioni software il tipo è l'operando della INT
   - Se sono eccezioni il tipo è implicito
2. Controlla il BIT P del gate è se = 0 allora significa che il gate non è definito e si genera una eccezione di protezione (tipo 11)
3. Confronta il *cs* del processore e indica il livello di privilegio corrente con DPL e se $cs < DPL$ allora genera una eccezione di protezione (tipo 13). Questo controllo è utile per far si che le interruzioni con INT usino tipi associati ad interruzioni software e non con tipi associati ad interruzioni esterne o eccezioni. Per far così il programmatore sistema può impostare $DPL = 0$ nei gate di interruzioni software e $DPL = 1$ nei gate di interruzioni esterne o eccezioni
4. Se $cs > L$ genera una eccezione (tipo 13). L'attraversamento di un gate può solo innalzare il livello di privilegio e non abbassarlo, questo perché se si attraversa un gate dopo l'esecuzione della routine associata si vuole poter tornare al programma principale e se si è a livello sistema non ci si può fidare di una routine utente perché potrebbe non ritornare mai.
5. Salva la pila in un registro apposito, mette RSP in SRSP
6. Se $cs \neq L$ allora significa che dobbiamo innalzare il privilegio e cambiare la pila in pila sistema. Quindi carica in RSP il puntatore alla pila sistema che si trova nel campo punt_nucleo nel TSS corrente che si trova nella GDT. Questo funziona nel nostro sistema in quanto prevediamo solo 2 livelli di privilegio. quindi se $cs = L$ significa già che siamo con la pila sistema. È importante il cambio pila per 2 motivi:
   (a) Il processore deve poter scrivere 5 parole lunghe senza sovrascrivere altre cose, e quindi non può fidarsi della pila utente
   (b) È bene che queste informazioni siano salvate in M1 in modo che l'utente non le possa corrompere, in particolare il contenuto del bit CS che indica il livello di privilegio da reimpostare una volta tornati dall'interruzione
7. Salva nella pila puntata da RSP 5 parole quadruple:
   - Una riservata alla segmentazione non significativa per noi
   - Il contenuto di SRSP (puntatore alla vecchia pila)
   - Il contenuto del registro CS (CPL)
   - Il contenuto di RFLAGS

- Il contenuto di RIP

(IMPORTANTE: RIP sta in cima)

8. Azzera TF per disattivare la modalità single-step
9. Se I/T = 1 allora pone IF = 0
10. Mette in RIP l'indirizzo della routine di interruzione

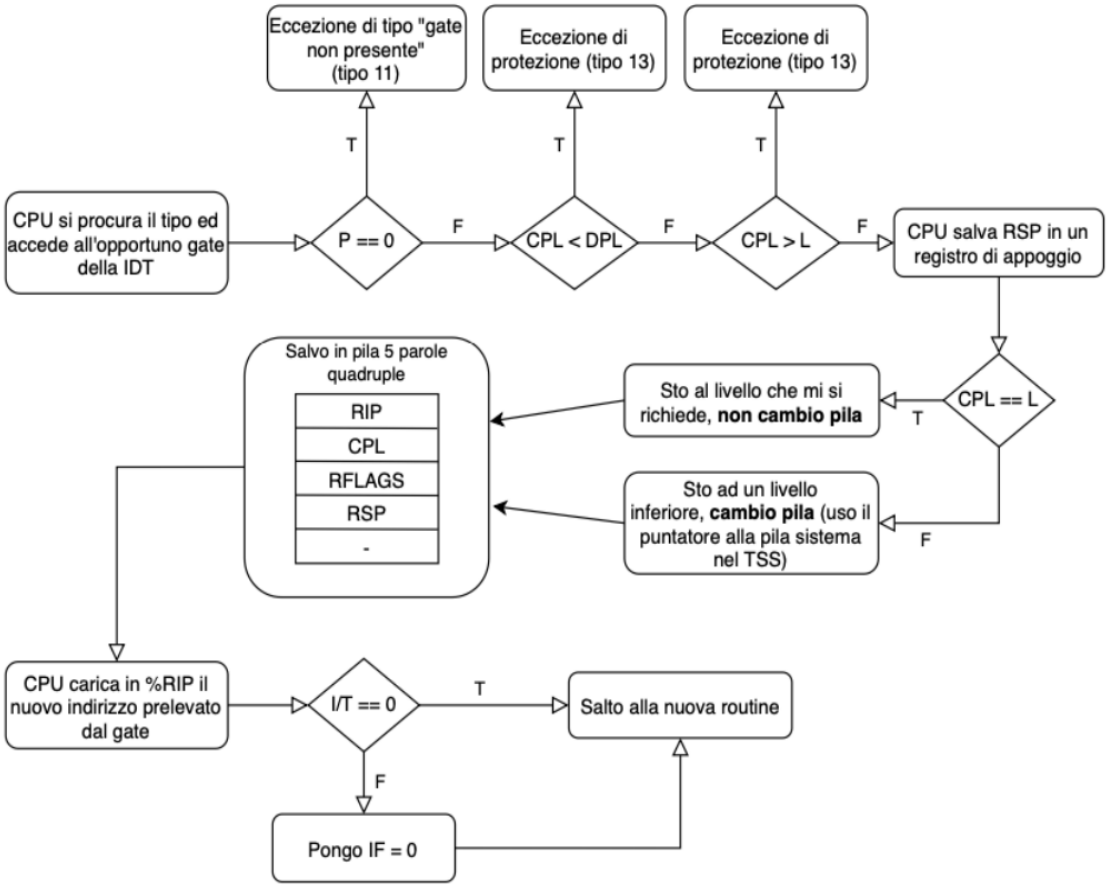


Figura 4: Flusso della CPU all'arrivo di una interruzione

2.10 Cosa sono le Routine?

Le routine sono parti programma fornite dal software base del sistema e vengono caricate in memoria durante l'inizializzazione dal programma di bootstrap che si occupate di allocare correttamente anche la IDT. Servono a gestire le interruzioni, vengono chiamate in seguito ad un attraversamento di un gate della IDT.

2.11 Come mai ogni routine deve terminare con IREQ?

Ogni Routine deve terminare con IRETQ in quanto svolge le operazioni opposte al meccanismo di interruzione infatti:

- Confronta il valore corrente di *cs* con quello salvato in pila e se quello salvato in pila è più alto genera una eccezione di protezione (La ireq può abbassare non innalzare il privilegio).
- Ripristina i valori di RIP, RFLAGS, RSP, CS usando i valori salvati in pila dal meccanismo di interruzione.

2.12 Cosa sono le Routine di Interruzione Esterne?

Le routine di interruzioni esterne si chiamano driver e in caso di interruzioni mascherabili e si occupano di gestire i trasferimenti e la comunicazione con i dispositivi e hanno un driver diverso per ogni interruzione. Le interruzioni non mascherabili invece che sono prodotte da cause catastrofiche come problemi di tensione hanno un driver unico che effettua le eventuali operazioni di recupero

2.13 Cosa sono le Routine di Interruzione Software?

Le routine di interruzione software o primitive di sistema sono quelle routine che il sistema mette a disposizione all'utente per fargli svolgere operazioni che sarebbe meglio non fargli svolgere in modo diretto. Ad esempio il cambio di un processo o la richiesta di trasferimento da o verso una periferica.

2.14 Cosa sono le Routine di Eccezioni?

Servono a recuperare il sistema in caso di errori e servono per gestirli e consentire il continuo del programma

- In caso di eccezioni trap e fault si gestisce l'errore e rimette in moto il programma
- In caso di abort si termina forzatamente il programma

2.15 Cosa sono i TSS e la GDT?

I TSS sono delle strutture dati in memoria che contengono informazioni utili tra cui il puntatore alla pila sistema. In origine erano usato per il cambio di processo via hardware, ne era quindi previsto uno per ogni processo. Nel nostro sistema invece non servono più se non per far funzionare il meccanismo delle interruzioni. Avremo quindi un solo TSS ma una pila sistema per ogni processo, quindi ad ogni cambio processo modificheremo il campo punt_nucleo. Per accedere al TSS bisogna accedere alla GDT(global descriptor table) puntata dal registro **gdtr**. Nel registro TR del processore troviamo l'indirizzo relativo alla GDT del TSS corrente. Le entrate relative al TSS contengono l'indirizzo base e la grandezza in byte. Con le istruzioni **ltr** e **lgdtr** si può caricare il registro tr e gdtr. Entrambe le istruzioni sono privilegiate e sia la GDT che tutte le TSS si devono trovare in memoria M1

2.16 A cosa serve il flag IOPL?

Il flag IOPL serve per dare consentire il privilegio nello spazio di IO. Consente di usare le istruzioni *in*, *out*, *cli* e *sti* che normalmente sono privilegiate.

2.17 Cos'è l'APIC?

L'APIC è un controllore delle interruzioni, quindi una componente hardware esterna alla CPU. L'APIC si occupa della gestione delle interruzioni, in particolare sceglie quali tra quelle in arrivo ha precedenza. Grazie all'APIC possiamo lasciare alla CPU un solo piedino /INTR per le richieste di interruzioni.

2.18 Disegno dell'APIC

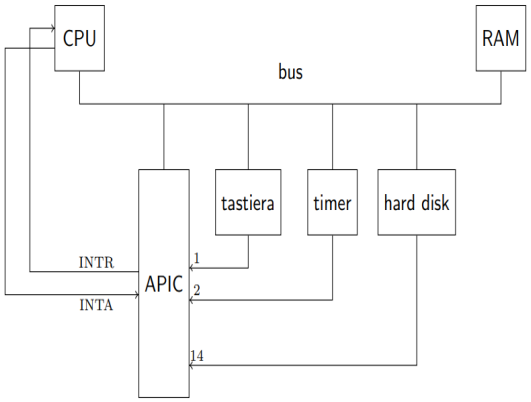


Figura 5: Controllore APIC

2.19 Quanti piedini ha l'APIC?

L'APIC ha 24 piedini da 0 a 23 a cui possono essere collegate le linee di interruzione di altrettante interfacce.

2.20 Come funziona l'handshake tra CPU e APIC?

1. Quando l'APIC vuole inviare una richiesta di interruzione attva /INTR e aspetta che diventi attivo /INTA.
2. Dopo l'esecuzione di ogni istruzione se IT=1 la CPU controlla /INTR e se lo trova attivo attiva /INTA
3. L'APIC legge /INTA attivo e allora passa il tipo di interruzione al BUS tramite il filo /TP e disattiva /INTR
4. LA CPU appena legge /INTR disattivo legge il tipo di interruzione e disattiva /INTA

2.21 Che differenza c'è tra riconoscimento sul fronte e input sul livello?

Per ogni piedino dell'APIC può essere impostata la modalità di riconoscimento del segnale in ingresso. Per riconoscimento sul fronte si intende che il riconoscimento di una nuova richiesta deve avvenire sul fronte di salita del piedino corrispondente. Ad ogni fronte su i se il bit t di IRR non è attivo lo attiva. Se il bit t di IRR è attivo ma quello di ISR non lo è allora l'APIC inizia l'handshake con la CPU. Quando la CPU accetta la richiesta allora l'APIC invia il tipo t e disattiva il bit in IRR e attiva il bit in ISR. Se arrivasse un altro fronte su i verrebbe comunque registrato in IRR senza però far partire l'handshake in quando il bit in ISR è ancora attivo. Una eventuale terza richiesta però andrebbe persa.

Il riconoscimento sul livello invece si ha quando arriva un nuovo fronte su i e il bit corrispondente su ISR è 0 allora si registra una nuova richiesta se non già attiva. Se invece il bit di ISR è 1 allora l'APIC smette di osservare il piedino i e aspetta una EOI che rimetterà il bit t corrispondete di ISR a 0. Se dopo l'EOI trova di nuovo o ancora il piedino attivo allora registra una nuova richiesta.

La differenza sta quindi nel riconoscimento di una nuova richiesta, nel caso del fronte

2.22 Come fa l'APIC a capire la priorità di una routine?

Ad ogni interruzione è associato un tipo su 8bit, l'APIC interpreta i 4 bit più significativi come livello di priorità. I livelli sono 16 e il massimo livello di priorità è il 15 che solitamente appartiene al timer che non può essere rimandato

2.23 Come si gestisce il caso in cui arrivi la stessa richiesta di interruzione di una già accettata?

Se una richiesta di interruzione è stata accettata ma non è ancora arrivato l'EOI relativo allora significa che il bit t corrispondente in ISR è 1 e il bit t corrispondente in ISR è 0. Se arriva una nuova richiesta di interruzione dello stesso tipo a l'APIC setta ad 1 il bit t in ISR senza avviare l'handshake con il processore in quanto, nella normalità, viene avviato solo se il bit t in ISR è 0. Quindi quanto i bit t in ISR e IRR sono entrambi settati significa che ho 2 richieste, una accettata e una in coda. Una eventuale terza richiesta verrebbe persa.

2.24 Come fa l'APIC a gestire le richieste di interruzione annidate?

Dato che le interruzioni hanno una priorità ha senso implementare la gestione annidata delle interruzioni, quindi le interruzioni possono interrompere altre interruzioni. Questo viene gestito dall'APIC nel registro ISR infatti il registro è a 256 bit ed è permesso avere più bit settati ad 1 contemporaneamente. Quindi abbiamo più interruzioni attive contemporaneamente in ordine di priorità e quando arriva l'EOI l'APIC resetta il primo bit 1 partendo da sinistra del registro ISR. Guarda poi in IRR e se cerca la richiesta con più alta priorità, sia Pr, e la cerca anche in ISR, sia Ps, se Pr $>$ Ps allora avvia

l'handshake con il processore e si comporta normalmente. Se invece Pr $\leq$ Ps l'APIC non inoltra nessuna richiesta, questo comprende anche il caso particolare di richiesta in coda in quanto se la priorità è la stessa significa che sia ISR che in IRR il bit t è settato

2.25 Come funziona la condivisione delle richiesta di interruzione?

Ad uno stesso piedino APIC possono essere collegate più interfacce, quindi tutte le interruzioni mandate da quell'interfaccia faranno partire la stessa routine di servizio. Per fare questo si potrebbe mettere un OR in uscita alle interfacce, però bisogna che le interfacce siano dotate di un registro per capire se la richiesta è stata inoltrata oppure no. Poi nella routine di servizio tramite il registro nell'interfaccia bisogna capire quale tra le tante sia passata. Per fare questo è necessario che il piedino dell'APIC sia impostato per il riconoscimento sul livello e non sul fronte in quanto il fronte in un OR rende invisibili eventuali altre richiesta dalle altre interfacce sullo stesso piedino.

2.26 Cosa è la configurazione open-collector e wired-or?

Le interfacce che condividono lo stesso piedino nella realtà non sono collegate con un OR fisico ma utilizzano una configurazione con uscite open-collector, cioè uscite che hanno il livello logico zero ottenuto collegando l'uscita a massa e il livello logico uno collegando l'uscita in alta impedenza. In questo modo funziona come un OR logico interpretando le uscite come attive basse, infatti quando una uscita di una interfaccia è 0 l'uscita complessiva è zero se invece tutte le uscite sono 1 allora l'uscita complessiva è portata a livello alto da una resistenza. Per poter usare questa configurazione il piedino dell'APIC deve essere impostato su il riconoscimento sul livello basso.

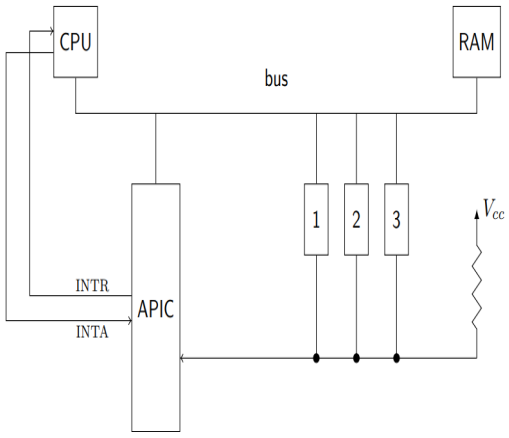


Figura 6: Controllore APIC open-collector

2.27 Quali azioni svolge il controllore APIC?

- Ogni volta che arriva una richiesta di interruzione sul piedino IR0_IR23 mette ad 1 il relativo di di IRR

    * Se il segnale è impostato sul fronte la richiesta è rappresentata da un segnale che da inattivo diventa attivo
    * Se il segnale è impostato sul livello la richiesta di interruzione è rappresentata da un segnale che è attivo nel momneto in cui l'APIC riceve l'EOI

- Svolge in sequenza queste azioni:

    1. Attende che il bit più significativo di valore 1 di IRR abbia priorità maggiore del bit più significativo di valore di ISR
    2. Invia alla CPU una richiesta di interruzione attivando /INTR e aspetta che la CPU attivi /INTA
    3. Quando riceve la risposta:
        (a) considera il bit BIT più significativo di valore 1 di IRR sia j-esimo;
        (b) pone il j-esimo bit di IRR a 0 e il j-esimo bit di ISR ad 1;
        (c) invia il tipo dell'interruzione j alla CPU tramite TP;
        (d) Rimuove la richiesta dalla CPU disattivando /INTR
    4. Ogni volta che arriva un EOI pone a 0 il bit più significativo di valore 1 di ISR

2.28 Nel meccanismo delle interruzioni, chi fa cosa?

PERIFERICA → Invia una richiesta di interruzione all'APIC

APIC → Sente la richiesta e la invia alla CPU

CPU → Dopo che finisce ogni istruzione corrente controlla se
(/INTR è attivo && IF == 1)?

- NO: FINE, continua normalmente
- SI: Attiva /INTA

APIC → invia il tipo al processore con TP

CPU →

- Riceve il tipo
- Lo usa per accedere alla IDT
- Aggiorna IF in base a I/T
- Salva RFLAGS
- Salta all'indirizzo scritto nel gate

- continua con il normale ciclo

SOFTWARE → Invia l'EOI

3 PROCESSI

3.1 Qual è la differenza tra un programma e un processo?

- Un programma può essere associato a più processi
- In generale non è solo il programma che decide come il processo si evolve nel tempo ma dipende anche dai dati in ingresso (ad esempio gli if di in programma dipendono dai dati e in base a quelli cambia il flusso del processo)
- Un processo può essere associato ad un solo programma

3.2 Quali sono gli stati di un processo?

- PRONTO, il processo è in attesa che venga schedulato per essere messo in esecuzione
- ESECUZIONE, il processo è il processo che attualmente ha il possesso della CPU
- BLOCCATO, il processo è in attesa che si verifichi un certo evento, una volta verificato questo evento si mette in stato pronto
- TERMINATO, il processo ha cessato la propria vita e non può più ri andare in esecuzione

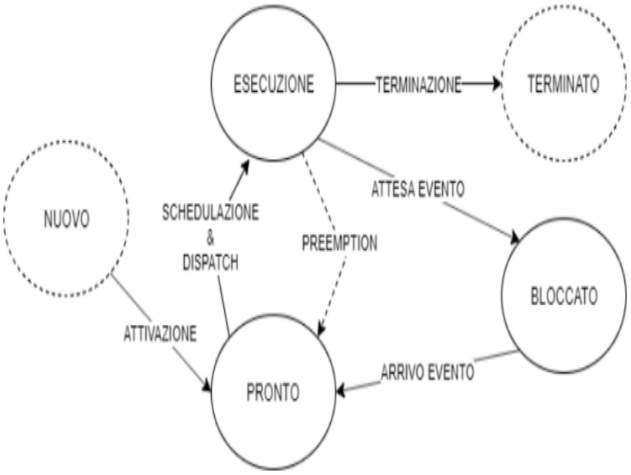


Figura 7: Stati di un processo

3.3 Quali sono le strategia adottate per la schedulazione?

- Schedulazione a divisione di tempo, la CPU assegna un tempo ad ogni processo trascorso il quale un timer interrompe la CPU causando preemption e schedulazione del prossimo processo pronto

- Schedulazione a priorità fissa: ad ogni processo è assegnata una priorità numerica al momento della creazione, in ogni istante il sistema deve garantire che sia in esecuzione il processo con priorità più elevata. si esegue una schedulazione quando:
  - un processo passa da esecuzione a bloccato, bisogna mettere in esecuzione il processo con priorità massima.
  - un processo passa da bloccato a pronto, potrebbe esserci un processo in stato pronto con priorità maggiore di quello in esecuzione, si deve quindi bloccare il processo in esecuzione e mettere in esecuzione quello con priorità superiore

3.4 Come sono realizzati i processi nel nucleo?

Nel nucleo le code dei processi sono implementate con delle liste, lista CODA , lista Pronti e lista ESECUZIONE ordinate per priorità, la lista esecuzione è una lista di un solo elemento. I processi sono implementati come descrittori di processo:

```
1 struct des_proc{
2     natl id; //identificatore
3     natw livello; //livello di privilegio Sistema o Utente
4     natl precedenza; //precedenza nelle code dei processi, priorità per la schedulazione
5     vaddr punt_nucleo;// puntatore alla base della pila sistema
6     naq contesto[N_REG]; //array contenente il contenuto dei registri generali del processo
7     paddr cr3; //radice del trie associato al processo
8     des_proc* puntatore; //puntatore al prossimo elementoi della lista
9 };
```

3.5 Com'è realizzato il cambio di processo nel nucleo?

Il cambio di processo può avvenire solo quando il processo in esecuzione si porta a livello sistema, quindi può avvenire per 3 ragioni:

- Il processo esegue una istruzione **int**
- Il processore genera una eccezione;
- Il processore accetta una interruzione esterna

In entrambi i casi il processore passa per un gate della IDT e si porta a livello sistema. Le routine a livello sistema hanno sempre questa forma

```
1 CALL salva_stato
2 ...
3 CALL carica_stato
4 IRETQ
```

la salva stato salva lo stato della CPU nel des_proc del processo identificato variabile esecuzione(in sistema.s), la carica_stato invece fa il meccanismo opposto, carica nella

CPU il contesto del processo entrante.
Il valore di RSP salvato in pila Sistema punta alla pila utente del processo ed è stato salvato con l'attraversamento del gate, mentre contesto[I_RSP] salvato dalla salva_stato è il contenuto di RSP dopo il salvataggio delle 5 parole quadruple e punta alla cima della pila sistema.
Quando la CPU accetta una interruzione smette di eseguire le istruzioni del processo attualmente in esecuzione e passa ad eseguire una routine di interruzione che usa le risorse del processo che si trova in esecuzione.

3.6 Cosa fa la salva_stato?

La salva stato si occupa di salvare il contesto del processo nel descrittore di processo attualmente in esecuzione. La salva_stato sporca due registri RBX e RAX come registri puntatore e di lavoro, infatti li salva tramite pushq e popq all'inizio e alla fine del programma. La salva stato deve salvare lo stato in cui si trova il processo prima di chiamare la funzione stessa, infatti bisogna prestare attenzione a che valore del registro RSP, che punta alla pila, bisogna salvare. In particolare nella salva stato abbiamo salvato in pila 8 byte con la CALL che contengono l'indirizzo di ritorno e 16 byte con le pushq di rbx e rax. Quindi dobbiamo salvare un valore della pila di 24byte sotto aggiungendo 24 al volore di RSP prima di salvarlo nell'array contesto. Considerando che lo stato in cui si trova il processo è subito successivo all'attraversamento del gate e quindi in pila sistema ho salvato le 5 parole quadruple e in cima avrò il valore del registro RIP, quindi l'RSP salvato nel contesto **punta al valore di RIP salvato dal meccanismo delle interruzioni**.

```
1  CALL salva_stato
2  ...
3  CALL carica_stato
4  IRETQ
5  ...
6  salva_stato:
7          pushq %rbx
8          pushq %rax
9
10         movq esecuzione, %rbx
11         movq %rbx, esecuzione_precedente
12
13         movq (%rsp), %rax
14         movq %rax, RAX(%rbx)
15         // usiamo %rax come appoggio per copiare il vecchio %rbx
16         movq 8(%rsp), %rax
17         movq %rax, RBX(%rbx)
18
19         movq %rcx, RCX(%rbx)
20         movq %rdx, RDX(%rbx)
```

```
21
22          movq %rsp, %rax
23          addq $24, %rax
24
25          movq %rax, RSP(%rbx)
26          movq %rbp, RBP(%rbx)
27          movq %rsi, RSI(%rbx)
28          movq %rdi, RDI(%rbx)
29          movq %r8,  R8 (%rbx)
30          movq %r9,  R9 (%rbx)
31          movq %r10, R10(%rbx)
32          movq %r11, R11(%rbx)
33          movq %r12, R12(%rbx)
34          movq %r13, R13(%rbx)
35          movq %r14, R14(%rbx)
36          movq %r15, R15(%rbx)
37
38          popq %rax
39          popq %rbx
40          ret
```

3.7 Cosa fa la carica_stato?

La carica stato fa il lavoro opposto della salva stato, quindi deve mettere nella CPU i valori dei registri che trova salvati nel descrittore di processo. La carica stato deve cambiare il valore di rsp e mettere la pila del processo che sta venendo caricato, il problema è che siamo dentro una CALL e quindi nella pila io ho l'indirizzo di ritorno. Con una popq prelevo l'indirizzo di ritorno nella pila(riga 5) e lo salvo in rcx e dopo aver cambiato pila(riga 17) mettendo il valore di rsp salvato nel contesto in rsp, metto nella nuova pila l'indirizzo di ritorno della CALL (riga 18)in modo che la retq alla fine mi porti dopo la chiamata della carica stato.
Un'altra cosa che la carica stato deve fare è aggiornare il campo punt_nucleo del TSS(riga 26)
La carica stato svolge anche un altro compito: nel caso in cui la primitiva invocata sia la terminate_p e quindi il processo deve essere terminato ma la terminate_p non distrugge la pila in quanto ci serve per poter lavorare. Una volta cambiata la pila nella carica stato, se il processo era terminato o abortito allora bisogna occuparsi di distruggere la pila sistema precedente(riga 22)

```
1  carica_stato:
2
3          movq esecuzione, %rbx //dispatch
4
5          popq %rcx   //ind di ritorno, va messo nella nuova pila
6
```

```
7           // nuovo valore per cr3
8           movq CR3(%rbx), %r10
9           movq %cr3, %rax
10          cmpq %rax, %r10
11          je 1f                       // evitiamo di invalidare il TLB
12                                  // se cr3 non cambia
13          movq %r10, %rax
14          movq %rax, %cr3                 // i  TLB viene invalidato
15  1:
16
17          movq RSP(%rbx), %rsp    //cambiamo pila
18          pushq %rcx              //mettiamo l'indirizzo di ritorno nella nuova pila
19
20          cmpq $0, ultimo_terminato
21          je 1f
22          call distruggi_pila_precedente
23  1:
24
25          movq PUNT_NUCLEO(%rbx), %rcx
26          movq %rcx, tss_punt_nucleo
27
28          movq RCX(%rbx), %rcx
29          movq RDI(%rbx), %rdi
30          movq RSI(%rbx), %rsi
31          movq RBP(%rbx), %rbp
32          movq RDX(%rbx), %rdx
33          movq RAX(%rbx), %rax
34          movq R8(%rbx), %r8
35          movq R9(%rbx), %r9
36          movq R10(%rbx), %r10
37          movq R11(%rbx), %r11
38          movq R12(%rbx), %r12
39          movq R13(%rbx), %r13
40          movq R14(%rbx), %r14
41          movq R15(%rbx), %r15
42          movq RBX(%rbx), %rbx
43
44          retq
```

3.8 Come funziona la creazione di un processo?

La creazione di un processo avviene con una chiamata alla activate_p(void f(natq), natq a, natl prio, natl liv) che è una primitiva di sistema raggiungibile tramite l'attraversamento di un gate. La primitiva controlla che che i dati forniti dall'utente siano corretti e poi tramite la funzione crea_processo() crea il processo. La funzione prende in ingresso la

priorità del processo da creare, il livello una funzione che dovrà essere il punto di partenza del processo una volta messo in esecuzione e il parametro "a" da passare alla funzione f. La activate_p() dopo la creazione del processo lo mette in lista pronti e incrementa il numero di processi. Poi restituisce l'id del processo.

La crea_processo(void f(natq), natq a, int prio, char liv) invece crea il processo vero e proprio e si occupa di creare tutte le sue strutture dati:

- Crea un nuovo descrittore e ci mette tutti i dati che gli passa dal activate_p().
- Mette in RDI il parametro "a" in modo che quando il processo parte la funzione f lo avrà come parametro
- Si occupa poi di allocare il root del trie e la pila sistema
- Simula una INT in modo tale che quando poi il processo verrà scelto dalla coda pronti e viene chiamata la carica stato e la ireq che dovranno trovare le cose in pila. Simuliamo un INT mettendo in pila sistema 5 parole quadruple, RIP in cima, CS, RFLAGS tutti a 0 tranne IF, RSP, e SS(non importante). In RSP non mette il valore di RSP ma mette l'indirizzo che avrà la pila utente
- Crea la pila utente

3.9 Come si invocano le primitive?

Il programmatore vuole permettere all'utente di invocare primitiva, per farlo però vogliamo che passi attraverso un gate. Per passare attraverso un gate via software serve una istruzione INT. Il c++ però non conosce l'istruzione INT, e noi vogliamo che gli utenti invochino la primitiva per esempio in questo modo:

```

void mioproc(natq a){
    ...
}
void main(){
    activate_p(mioproc, 10, 10, LIV_UTENTE);
    ...
}
```

La chiamata viene tradotta con una CALL e non con una INT, quindi noi forniamo agli utenti delle funzioni in utente.s dove per ogni primitiva fornita ci sta una INT$tipo per attraversare il gate.

```

activate_p:
    int $TIPO_A
    ret

```

I parametri invece rimando nei registri adibiti allo scopo, infatti la funzione in utente.s non li tocca e attraversato il gate e cambiata la pila parte la salva stato che sta molto attenta a non sporcare quei registri, quindi i parametri sono rimasti li e arrivano tranquillamente alla routin in c++.

3.10 L'utente può in qualche modo accedere al modulo sistema e scavalcare la INT?

Anche sapendo gli indirizzi di dove sono salvati dati che mi servono, da utente non potrei comunque accedervi perché la protezione è fatta in hardware e quindi io utente non posso in nessun modo accedere ad indirizzi dedicati al sistema, in quanto la MMU non mi darà accesso.

3.11 Come si restituisce un valore in una primitiva e perché?

In una primitiva non posso fare semplicemente return valore; per poter restituire qualcosa. Questo il return mette in RAX del processo re il valore da restituire ma poi la carica stato sovrascrive quel valore. Quindi metto il valore da restituire dentro all'RAX salvato nel contesto. $p \rightarrow contesto[I\_RAX] = valore$;

3.12 Perché è stato introdotto il meccanismo dei semafori?

Nel sistema multiprocesso il controllo della CPU può saltare da un processo all'altro senza che l'utente possa controllarlo. I processi inoltre hanno anche una memoria condivisa alla quale possono accedere per leggere o scrivere valori. Bisogna quindi risolvere 2 problemi principali, quello della mutua esclusione e della sincronizzazione.

3.13 Cosa è e come si risolve il problema della mutua esclusione?

Il problema consiste quando dei processi usano dati condivisi con altri processi e può capitare che durante l'esecuzione di un processo, un altro prenda il controllo della CPU modificando quei dati e quindi rendendoli inconsistenti per il primo processo. Bisogna implementare quindi la Mutua Esclusione cioè rendere accessibili quelle strutture dati ad un processo per volta. Nel nostro sistema è risolta con l'introduzione dei semafori.

3.14 Cosa è e come si risolve il problema della sincronizzazione?

Con la sincronizzazione invece si vuole gestire l'ordine di accesso ad una determinata struttura dati, ad esempio voglio che il processo A esegua una azione sempre dopo un processo B. Nel nostro sistema è implementata con l'introduzione dei semafori.

3.15 come sono implementati i semafori?

Sono implementati con una struttura des_sem e un array di descrittori di semafori in cui la prima metà sono dedicati agli utenti e l'altra metà al sistema. I semafori si trovano nel modulo sistema quindi le strutture dati sono inaccessibili all'utente se non con l'utilizzo

di routine di sistema.
L'idea consiste nell'avere un semaforo che contiene dei gettoni e consente ad un processo di superarlo sono se all'interno c'è almeno un gettone, altrimenti si blocca e si mette in una coda all'interno del semaforo. Quindi la nostra struttura dati prevede un contatore di gettoni e una lista di processi ordinata per priorità. Quando il campo counter è negativo il suo modulo indica il numero di processi in coda.

```

des_sem array_dess[MAX_SEM * 2];

natl sem_allocati_utente  = 0;
natl sem_allocati_sistema = 0;

struct des_sem{
    int counter;
    des_proc* pointer;
}
```

3.16 Come si usano i semafori?

Per poter usare i semafori ci sono delle apposite routine di sistema, quelle più utili sono: sem_ini che inizializza un semaforo, sem_wait che preleva un gettone sem_signal che mette un gettone.

3.17 Cosa fa la sem_ini?

La sem_ini semplicemente si occupa di allocare un semaforo se possibile e restituire l'indice in cui è stato salvato il semaforo nell'array. Dell'allocazione se ne occupa la alloca_sem() che semplicemente controlla il livello del processo per vedere se inserire nella prima metà o nella seconda metà dell'array il nuovo semaforo, se non c'è posto restituisce -1;

```
extern "C" void c_sem_ini(int val)
{
        natl i = alloca_sem();

        if (i != 0xFFFFFFFF)
                array_dess[i].counter = val;

        esecuzione->contesto[I_RAX] = i;
}

```

3.18 Cosa fa la sem_wait?

Preleva un gettone dal semaforo se non ci sono gettoni mette in processo in lista pronti e chiama lo schedulatore.

```
extern "C" void c_sem_wait(natl sem)
{
        if (!sem_valido(sem)) {
                flog(LOG_WARN, "semaforo errato: %d", sem);
                c_abort_p();
                return;
        }
        des_sem *s = &array_dess[sem];
        s->counter--;

        if (s->counter < 0) {
                inserimento_lista(s->pointer, esecuzione);
                schedulatore();
        }
}
```

3.19 Cosa fa la sem_signal?

Il processo inserisce un nuovo gettone nel semaforo e se ci sono altri processi bloccati nel semaforo rimuove quello a priorità più alta e lo mette in lista pronti fa preeption e si mette in lista pronti e poi chiama lo schedulatore.

```
extern "C" void c_sem_signal(natl sem)
{
        // una primitiva non deve mai fidarsi dei parametri
        if (!sem_valido(sem)) {
                flog(LOG_WARN, "semaforo errato: %d", sem);
                c_abort_p();
                return;
        }

        des_sem *s = &array_dess[sem];
        s->counter++;

        if (s->counter <= 0) {
                des_proc* lavoro = rimozione_lista(s->pointer);
                inspronti();        // preemption
                inserimento_lista(pronti, lavoro);
                schedulatore();        // preemption
        }
}
```

3.20 Com'è implementata la mutua esclusione?

La mutua esclusione è implementata con un semaforo con all'interno un solo gettone, in modo da far entrare un solo processo alla volta.

```
natl mutex = sem_ini();
...
sem_wait(mutex);
...//azioni da svolgere in mutua esclusione
sem_signal(mutex);
```

3.21 Come è implementata la sincronizzazione?

La sincronizzazione è implementata con semaforo inizializzato con 0 gettoni. Siano 2 processi P_a e P_b che svolgono rispettivamente le azioni A e B, voglio che B venga svolta sempre dopo A

```
        natl sync = sem_ini(0);

Pa:     A;
        sem_signalsync);
        ...
        ...
        ...
Pb:     sem_wait(sync);
        B;
```

In questo modo se P_a non ha svolto A ma P_b cerca di svolgere B, nel semaforo non ci sono gettoni e rimane bloccato in attesa che P_a invochi la sem_wait per inserire un gettone. Di contro se P_a svolge A e inserisce un gettone allora P_b non rimane bloccato.

4 MEMORIA VIRTUALE

4.1 Perché è stata introdotta?

Nel sistema multiprocesso in teoria ogni volta che cambia un processo le strutture dati del processo uscente vengono ricopiate nello swap (parte dell'hard disk) e quelle del processo entrante ricopiate in M2. Questo meccanismo è lento ma anche efficiente dal punto di vista della protezione in quanto il processo in esecuzione non può accedere in nessun modo alla memoria degli altri processi.
Si vuole evitare questo continuo spostamento tra RAM e swap mantenendo però le seguenti proprietà:

- Isolamento tra i processi, ciascun processo può accedere solo alla propria memoria privata, non anche a quella degli altri processi.
- Semplicità nel collegamento, il collegatore può assumere che ogni programma ha accesso a tutta la memoria M2
- Semplicità nel caricamento o scaricamento dei registri, la memoria di ogni processo viene caricata e scaricata sempre agli stessi indirizzi.
- Condivisione della memoria, il sistema può evitare di spostare parti di memoria condivise tra i processi durante un cambio di processo.

L'idea è quindi quella di far credere ad ogni processo di avere a disposizione la CPU e la memoria M2 tutta per se quando invece ha una memoria virtuale e una CPU virtuale. Quindi durante l'esecuzione di un processo si useranno indirizzi virtuali che verranno poi tradotti in indirizzi fisici dall'MMU.

4.2 Come funziona il meccanismo della paginazione?

L'idea è quella di avere tutti gli indirizzi generabili da un processo in regioni naturali dette *pagine* e applicare una traduzione diversa per ogni pagina, la memoria fisica viene anche lei suddivisa in regioni naturali dette *frame*. Questa traduzione avviene tramite l'MMU che traduce gli indirizzi virtuali in indirizzi fisici.

4.3 Da cosa è composto un indirizzo virtuale e uno fisico?

Un indirizzo virtuale è composto da 48 bit (i restanti fino a 64 sono normalizzati) i primi 12 meno significativi sono l'offset mentre i restanti 36 sono il numero di pagina $v = (p, o)$
Un indirizzo virtuale invece è composto da 52 bit, i 12 meno significativi sono l'offset mentre i restanti 40 sono detti numero di frame $f = (n, o)$

4.4 Come viene usato il TRIE per il meccanismo della paginazione?

La traduzione degli indirizzi virtuali avviene utilizzando una struttura a TRIE. Nel nostro sistema usiamo pagine e offset di 4KiB. Un numero di pagina è composto da 36 bit, vengono divisi in 4 gruppi da 9 bit. Ogni nodo del TRIE conterrà una tabella di $2^9 = 512$ entrate con puntatori al nodo successivo mentre le foglie dell'albero conterranno le traduzioni degli indirizzi fisici.

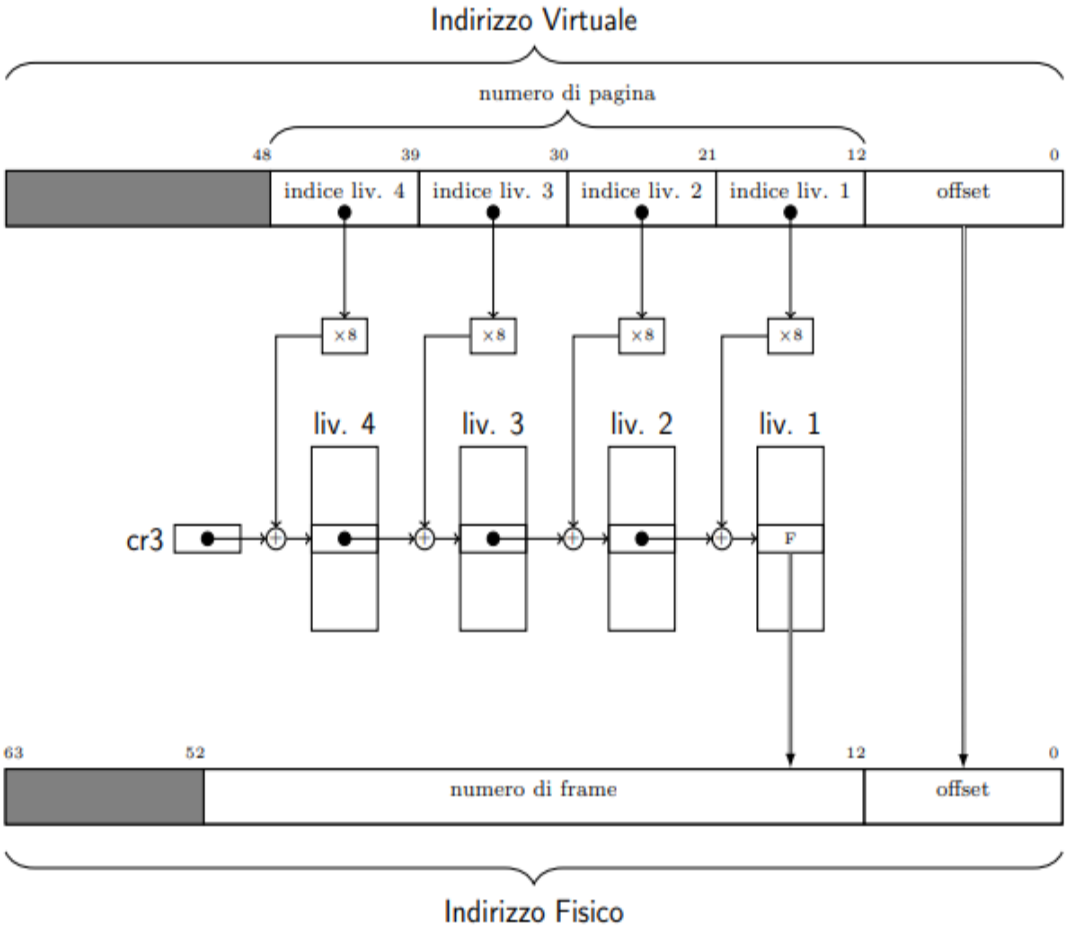


Figura 8: TRIE-MMU

Inoltre i descrittori delle varie tabelle contengono dei flag che permettono all'MMU di svolgere funzioni di protezione e altri meccanismi.

4.5 Come sono fatti i descrittori di tabella?

I descrittori di tabella sono indirizzi fisici che indicano dove si trova la tabella in memoria e quelli di Liv 2, 3, 4 sono fatti così:

- BIT PS (Page Size) permette di avere pagine di dimensioni maggiori di 4KiB

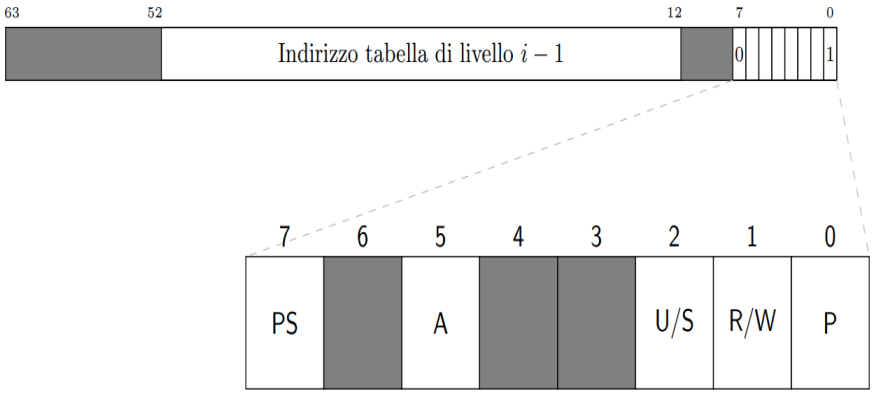


Figura 9: Descrittore di tabella di liv I

- BIT A (Access) serve per fare statistiche degli accessi in memoria, quando la MMU passa setta i BIT A ad 1
- BIT U/S (User/System) se 1 allora è accessibile a livello utente, altrimenti solo sistema
- BIT R/W (Read/Write) se 1 allora si può scrivere, altrimenti sola lettura
- BIT P (Present) se 1 allora l'entità è presente in memoria fisica

I descrittori di tabelle di Livello 1 invece sono dei descrittori di frame e sono fatti così:

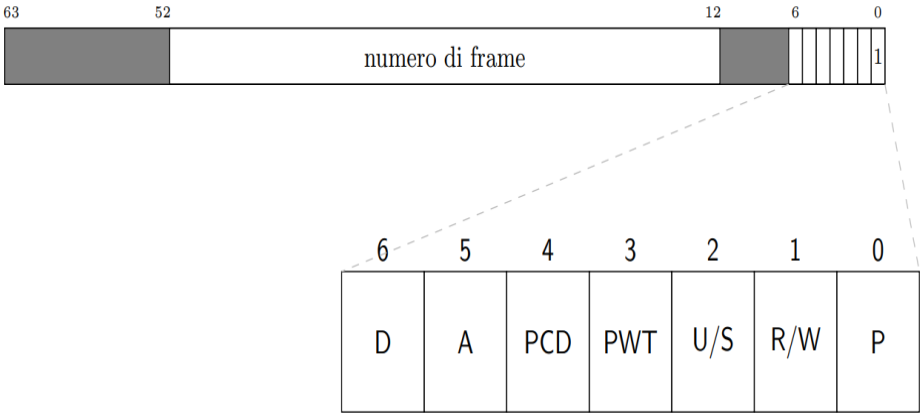


Figura 10: Descrittore di tabella di liv i

- BIT D (Dirty) se 1 indica se in quell'indirizzo sono avvenute operazioni di scrittura e serve per quando il sistema deve togliere una pagina dalla memoria fisica per metterla nello swap.
- BIT A
- BIT PCD(Page Cache Disabled) se 1 allora la cache viene disabilitata

- BIT PWT(Page Write Thought) ordina alla cache di usare la politica Write Thought
- BIT U/S
- BIT R/W
- BIT P

Se il BIT D:

- D=0, la pagina può essere rimpiazzata senza fare swap-out perché la copia nello swap è valida
- D=1, la pagina deve essere ricopiata prima di essere rimpiazzata perché quella nello swap non è più valida.

Se il BIT PCD:

- PCD=1, allora PWT viene ignorato e la cache non intercetta più le operazioni, la routine di inizializzazione pone a PCD=1 in tutti gli indirizzi mappati nello spazio di I/O
- PCD=0, allora se PWT=1 allora la cache quando intercetta mette il dato anche in memoria. Nel caso della Memoria Video il BIT PWT=1 perché il dato deve sempre arrivare in memoria affinchè il controllore possa rappresentarlo sul display

I flag in comune tra i descrittori di tabella e quelli di frame per essere validi devono essere settati in tutti livelli. Ad esempio se il bit U/S sta in tutti i descrittori ad 1 tranne che nel descrittore di tabella di livello 2 allora l'indirizzo finale non è accessibile all'utente come se fosse stato 0 in tutti quanti.

4.6 Chi modifica i questi flag?

I flag PCD PWT US RW e P sono scritti dal software mentre quelli D e A sono scritti dall'MMU.

4.7 Cosa fa l'MMU?

La MMU intercetta gli indirizzi virtuali generati dalla CPU e li traduce tramite un TRIE a 4 livelli in cui l'ultimo livello contiene l'indirizzo virtuale. Dati i bit 12-47 del numero di pagina e il registro CR3 che punta al TRIE del processo attualmente in esecuzione:

- Prende i bit 39-47 i_4 li moltiplica per 8. Concatena a CR3: $cr3 + i_4 \times 8$ e ottiene l'indirizzo del descrittore di livello 4.
- Concatena il risultato ai bit 30-38 i_3 e altri 3 bit a 0 e ottiene il descrittore di liv 3
- Concatena il risultato ai bit 21-29 i_2 e altri 3 bit a 0 e ottiene il descrittore di liv 2

- Concatena il risultato ai bit 12-20 i_1 e altri 3 bit a 0 e ottiene il descrittore di liv 1
- Estrae dal descrittore di livello 1 il numero di frame che contiene la pagina
- Concatena il numero di frame con l'offset e ottiene l'indirizzo fisico che è la traduzione dell'indirizzo virtuale di partenza

L'MMU lavora anche con i flag dei vari descrittori di pagina in particolare:

- Passa al controllore cache le informazioni PCD e PWT trovati nel descrittore di livello 1
- Controlla i bit R/W e se sono settati in tutti e 4 i descrittori la scrittura è permessa
- Controlla i bit U/S e se sono settati in tutti e 4 i descrittori il frame è accessibile a livello utente
- Setta i BIT A in tutti i descrittori se non sono già settati
- Se l'operazione è di scrittura setta i BIT D in tutti i descrittori
- Se trova in un descrittore il BIT $PS = 1$ allora non continua ad andare più a fondo con i livelli e usa il risultato come indirizzo fisico, serve per usare pagine di grandi dimensioni. Se la MMU si ferma al descrittore di liv 2 perché ha trovato $PS = 1$ allora la pagina avrà dimensione 2MiB, liv 3 invece 1GiB

4.8 Che cosa è la finestra di memoria?

Si vuole poter fare in modo che le routine di sistema possano utilizzare direttamente indirizzi fisici, dato che la MMU non può essere disattivata si crea una Finestra sulla Memoria. In pratica si mappa con traduzioni identità tutta la memoria, poi si mettono queste traduzioni nello spazio di indirizzamento di ogni processo. È come se nella parte alta di ogni processo accessibile solo a livello sistema avessimo creato una finestra che permette di vedere tutta la memoria così com'è

4.9 Com'è implementata?

La finestra di memoria nel nostro nucleo viene implementata dalla funzione crea_finestra_FM()

```
1 bool crea_finestra_FM(paddr root_tab)
2 {
3     auto identity_map = [] (vaddr v) -> paddr { return v; };
4 
5     natq first_reg = dim_region(1);
6     if (map(root_tab,DIM_PAGINA,first_reg,BIT_RW,identity_map)!=first_reg)
7         return false;
8 
9     tab_iter it(root_tab, 0xb8000, DIM_PAGINA);
```

```
10      while (it.down())
11          ;
12      tab_entry& e = it.get_e();
13      e |= BIT_PWT;
14
15      if (MEM_TOT > first_reg) {
16        if(map(root_tab,first_reg,MEM_TOT,BIT_RW,identity_map,2)!=MEM_TOT)
17              return false;
18      }
19
20
21      vaddr        beg_pci = 4*GiB - 20*MiB,
22          end_pci = 4*GiB;
23      if(map(root_tab,beg_pci,end_pci,BIT_RW|BIT_PCD|BIT_PWT,identity_map,
24      2) != end_pci)
25          return false;
26
27      return true;
28  }
29
```

- Non si mappa la pagina 0
- Da DIM_PAGINA alla fine della prima regione si mappa la memoria video, poi tramite un iteratore si setta il BIT PWT.
- Da first_reg fino alla fine della Memoria si mappa tutto usando il quinto parametro della *map* che è il PS (Page Size) creando quindi pagine di 2MiB
- Si mappa infine la parte di memoria dedicata alle periferiche I/O con la politica di cache disattivata in quanto la cache non deve intercettare le operazioni di I/O

4.10 Cosa è e perché è stato introdotto il TLB?

Il TLB (Translation Lookaside Buffer), è una cache associata all'MMU per salvarsi le traduzioni di indirizzi virtuali. È stata introdotta perché la traduzione di un indirizzo virtuale richiede ben 5 accessi in memoria 4 per le tabelle uno per il dato, e quindi le prestazioni ne risentivano. Se invece una traduzione è stata salvata nel TLB ora per ottenere l'indirizzo fisico mi basta accedere li.

4.11 Disegno del TLB

4.12 Come funziona il TLB?

Il TLB funzione come una normale cache, solitamente è a n vie con algoritmo pseudo-LRU, Conserva il valore dei flag dei descrittori e se fa *miss* allora la MMU si comporta

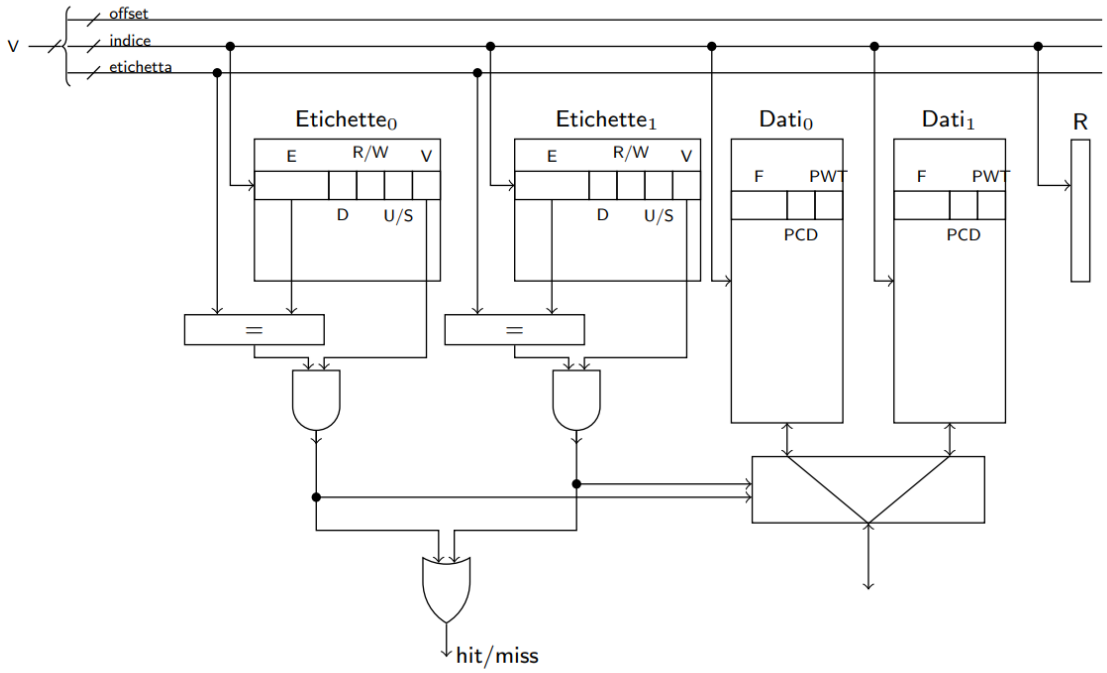


Figura 11: TLB a 2 vie

normalmente. La TLB come tutte le cache è trasparente al software e non è in nessun modo possibile leggerci dentro. Può però essere utile invalidare tutta o una parte della TLB in caso in cui le traduzioni non siano più valide, un modo per farlo è scrivendo una qualsiasi cosa, anche se stesso, in cr3.

4.13 Come si comporta il TLB per quanto riguarda i flag? E quali soluzioni adotta per i BIT A e D?

La MMU non si occupa solo delle traduzioni ma svolge alcuni meccanismi come quelli di protezione. Nel TLB allora ho bisogno di salvare i flag che mi permettono di poter continuare a far funzionare questi meccanismi. In particolare:

- Per i flag U/S e R/W basta salvare un AND di tutti i bit di tutte e 4 le tabelle di traduzione.
- Per i flag PCD e PWT vengono salvati insieme al descrittore di frame e passati all'MMU che si occuperà di passarli al controllore cache.

I flag A e B invece sono più problematici in quanto vengono aggiornati in fase di traduzione:
Per quanto riguarda il bit A la soluzione è quella di aggiornare l'albero solo se si ha una *miss* nel TLB. Il problema sussiste quando il software per qualche motivo decide di azzerarli nelle tabelle. In quel caso la soluzione deve essere in software e quando vengono azzerati i bit A allora bisogna invalidare in parte o del tutto il TLB. Per il BIT D invece

non basta, la soluzione non può essere fatta in software e quindi è fatta totalmente in hardware. La prima volta che viene salvato una traduzione nel TLB il bit D viene messo a 0, ora si fa in modo che se:

- $D = 0$ e ho un accesso in scrittura, il TLB fa *miss* e la MMU scorre tutto l'albero e la traduzione viene rimessa nel TLB ma stavolta con $D = 1$
- $D = 1$ la MMU prende la traduzione dal TLB

Questo perché se la pagina viene scelta come vittima per essere eliminata dalla memoria se trovo $D = 0$ allora non si farebbe *swap-out*

4.14 Come si funziona il TLB per le pagine di grandi dimensioni?

Il TLB funziona solo con un tipo di formato, se le pagine sono di dimensioni 4KiB non posso salvare quelle di 2MiB o superiori. Una soluzione adottata era quella di salvare direttamente tutti e le 512 traduzioni della tabella di livello superiore, nel caso 4KiB quelle della tabella di Liv 2, questo mi farebbe occupare 512 posizioni del TLB. Ora semplicemente si hanno TLB per ogni formato e dato che il TLB non può sapere a priori la grandezza della pagina prima della ricerca, la ricerca viene fatta in parallelo e quindi non si perde nemmeno in prestazioni.

5 BUS PCI

5.1 Cosa è lo standard PCI?

Lo Standard PCI è uno standard introdotto da INTEL che stabilisce quali sono i collegamenti del BUS e il protocollo che le schede devono usare per comunicare. Definisce inoltre 3 spazi di indirizzamento:

- Spazio di memoria
- Spazio di I/O
- Spazio di configurazione

Nello spazio di configurazione ci sono dei registri che ogni periferica deve avere per rispettare lo standard. Il software di avvio può accedere allo spazio di configurazione e scoprire quali periferiche sono collegate al BUS e come sono configurate.
Lo standard PCI non è legato al un tipo di processore ma prevedere un dispositivo *ponte ospite PCI* che fa da tramite con il bus standard del processore e il bus PCI.
Lo Standard prevede inoltre non un solo BUS ma un albero di al massimo 256 BUS collegati al BUS PCI con ponti PCI-PCI. Ad ogni BUS si possono collegare fino a 32 dispositivi e ogni dispositivo può avere fino ad 8 funzioni.

5.2 Quali sono i principali collegamenti del BUS PCI?

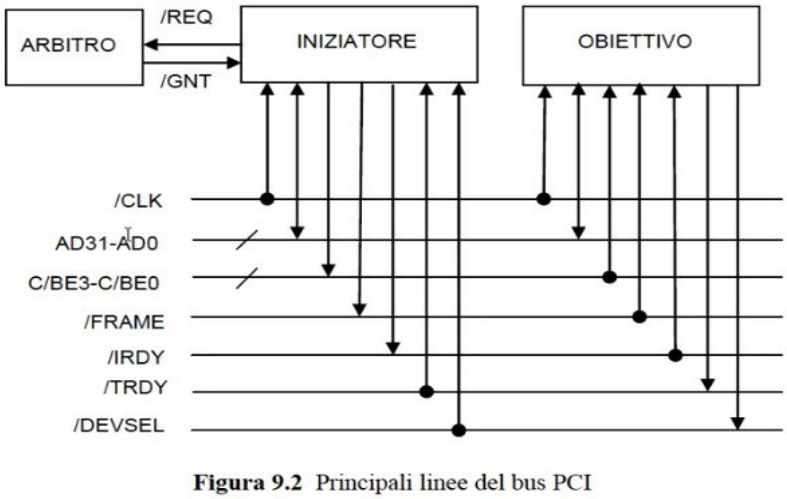


Figura 9.2 Principali linee del bus PCI

Figura 12: Collegamenti BUS PCI

- REQ e GNT: handshake iniziatore-arbitro per iniziare una transazione sul BUS
- CLK: Non è il clock del processore ma un clock più lento

- AD[31:0]: Adress o Data, in ingresso/uscita per tutti i dispositivi. Possono contenere indirizzi o dati. Quando contengono indirizzi devono essere multipli di 4 quindi A1A0 = 00;
- C/BE[3:0]: (Command o Byte enabler) Uscita dall'iniziatore e ingresso all'obbiettivo. Durante la fase di indirizzamento fanno da linee di comando, cioè indicano il tipo di operazione. Durante la fase di dati invece fanno da byte enable.
- FRAME: uscita iniziatore, ingresso obbiettivo. Filo di controllo che determina l'inizio e la fine di ogni transazione.
- IRDY e TRDY fili di hadshake per la fase di dati
- DEVSEL: uscita obbiettivo, viene attivato quando l'obbiettivo riconosce uno dei sui indirizzi durante la fase di indirizzamento

5.3 Come funzionano le operazioni di letture e scrittura?

Il protocollo è il seguente:

1. FASE di ARBITRAGGIO:
   - Iniziatore attiva **/REQ**
   - L'arbitro prima o poi attiva **/GNT**
   - L'iniziatore aspetta che **/FRAME e /IRDY** siano disattivati, altrimenti significa che ci sta una transazione ancora in corso sul BUS
2. FASE DI INDIRIZZAMENTO:
   Il dispositivo iniziatore:
   - Pilota **AD** come address
   - Pilota **C/BE** come command cioè indica se R/W e in che spazio di indirizzamento(memoria, IO, configurazione)
   - Attiva **/FRAME** dando inizio alla transazione

   Se /DEVSEL non si attiva in 6 cicli di clock allora si assume che nessun dispositivo risponda a quegli indirizzi e si termina la transazione
3. FASE DI DATI(possono essere tante)
   L'iniziatore:
   - Pilota le linee **C/BE** come byte enabler
   - Se la transazione è nello spazio di memoria il prossimo indirizzo è ottenuto incrementando il precedente di 4
   - Se la transazione è nello spazio di IO l'incremento dell'indirizzo dipende dall'interfaccia

- Se l'operazione è in **SCRITTURA** l'**iniziatore** pilota **AD** come data, attiva **/IRDY** e attende che si attivi **/TRDY**
- Se l'operazione è in **LETTURA** l'**obbiettivo** pilota **AD** come data e attiva **/TRDY**, e attende che venga attivato **/IRDY**
- Nell'ultima fase di dati l'iniziatore non mantiene attivo **/FRAME**

5.4 Come funziona lo spazio di configurazione?

Lo Standard prevede uno spazio di configurazione di 64 parole quadruple per ogni funzione, in questo spazio ciascuna funzione rende accessibili i registri di configurazione

+3	+2	+1	+0	offset
Device ID		Vendor ID		0x00
Status		Command		0x04
Class Code			Revision ID	0x08
...	Header Type	...	Cache Line Size	0x0c
Base Address Register 0				0x10
Base Address Register 1				0x14
Base Address Register 2				0x18
Base Address Register 3				0x1c
Base Address Register 4				0x20
Base Address Register 5				0x24
...				0x28
...		...		0x2c
...				0x30
...			...	0x34
...				0x38
...	...	Intr. Pin	Intr. Line	0x3c

Figura 13: Spazio di configurazione di ogni funzione

I dispositivi non sono obbligati a mettere tutti i registri accessibili nello spazio di configurazione. Quelli obbligatori sono:

- **VENDOR ID** (lettura) identifica il produttore (8086 intel)
- **DEVICE ID**(lettura) identifica il la funzione

- **COMMAND**(lettura/scrittura) può essere 0, 1, 2. Se 0 e 1 abilita a rispondere alle transazioni rispettivamente in memoria e IO. Se 2 attiva la funzionalità di BUS MASTERING
- **STATUS**(lettura) descrive alcune capacità della funzione e lo stato di acluni eventi legati a situazioni di errore
- **CLASS CODA**(lettura) identifica il tipo della funzione
- **REVISION ID**(lettura) numero di revisione della funzione
- **HEADER TYPE**(lettura) deve essere 0 per tutti i ponti che non siano PCI-PCI o PCI-CardBus

Intr. Pin e Intr. Line invece indicano rispettivamente a quali pin di interruzione è collegata la funzione e quali piedini dell'APIC.

5.5 Come funzionano le transazione nello spazio di configurazione?

Dal punto di vista hardware ogni PCI tranne il ponte-PCI possiede un ingresso IDSEL. Il punte-PCI invece possiede un collegamento per ognuno di questi ingressi. Una transazione verso lo spazio di configurazione avviene quindi similmente alle altre solo che il ponte-PCI attiva anche l'IDSEL corrispondente e il dispositivo presente in quell'ingresso risponde attivando /DEVSEL, per il resto è uguale alle altre transazioni. Lo standard impone che se /DEVSEL non viene attivato entro un tot di cicloi di clock viene il ponte riconosce quello slot come vuoto.

Dal punto di vista software la CPU non possiede istruzioni specifiche ma il ponte-PCI mette a disposizione 2 registri:

- **CAP** (Configuration Address Port)
- **CDP** (Configuration Data Port)

Per identificare una funzione servono 3 numeri:

1. Il numero del BUS, nel nostro caso è sempre 0
2. Il numero del dispositivo, dipende dal piedino IDSEL
3. Il numero della funzione che viene scelto dal produttore

Questi 3 numeri e l'offset vanno scritti dentro il registro CAP, una volta fatto il ponte ospite-PCI tradurrà le scritture e le letture in CDP in equivalenti transazioni nello spazio di configurazione in particolare il ponte:

- Attiva l'uscita IDSEL che trova in CAP
- Copia il numero di funzione e l'offset in A[10:0]

- Usa come /BE[3:0] i primi 4 byte enable del processore

Il ponte se l'operazione è di lettura ma nessun dispositivo risponde attivando /DEVSEL restituisce un valore composto da tutti 1 0XFFFF.
Questo permette al software di rilevare se in una posizione c'è un dispositivo collegato oppure no. In particolare, dato che non ci sono Vendor ID del valore di 0XFFFF, basta inserire in CAP le informazioni per leggere il Vendor ID dei dispositivi al variare di tutti i possibili numeri di dispositivi(da 0 a 32), e per ognuno di tutte le funzioni(da 0 a 7) e se come risultato si ottiene 0XFFFF vuol dire che non c'è nessun dispositivo collegato. Ovviamente se in un dispositivo non è presente la funzione 0 non ha senso controllare le altre.

5.6 Cosa sono i Base Address Register?

Il Software di inizializzazione ha l'obbiettivo di assegnare ai dispositivi indirizzi univoci negli spazi di memoria e IO, in modo tale che poi il software possa accedere alla periferica direttamente da li. Il dispositivo deve mettere a disposizione dei blocchi di indirizzi nei quali rendere disponibili i registri specifici della funzione. Il costruttore deve anche decidere per ogni blocco se deve trovarsi nello spazio di IO o di memoria e che configurazione all'interno del blocco avranno i registri (Ad esempio negli esercizi di IO i vari STS RBR e così via stanno tutti dentro ad un BAR, solitamente BAR0), quello che il costruttore non può decidere è l'indirizzo del blocco nello spazio di memoria o IO.
Ogni blocco ha b bit non scrivibili, il primo bit meno significativo se vale 1 significa che il BAR deve trovarsi sullo spazio di IO se invece vale 0 deve trovarsi nello spazio di memoria. Nel caso dello spazio di memoria $b \geq 4$,dimensione minima 16 byte, in quanto i bit 1-3 contengono informazioni sul tipo di memoria, nel caso di IO invece $b \geq 2$, dimensione minima 4 byte.

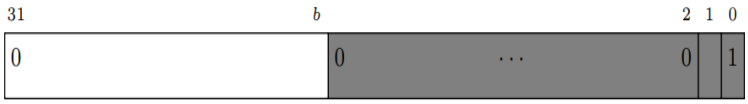


Figura 14: BAR IO

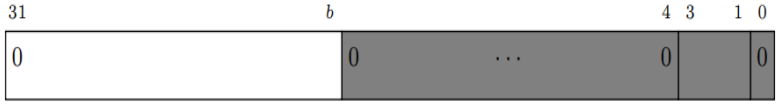


Figura 15: BAR memoria

Il software di configurazione deve quindi assegnare a questi blocchi delle regioni nello spazio di memoria o di IO, una volta assegnato lo inserisce nei bit scrivibili del blocco, in questo modo il BAR azzera i 2 o 4 bit meno significativi per ottenere l'indirizzo di

partenza del blocco.
Una volta inserito l'indirizzo nel blocco il software di inizializzazione deve inserire nel Command Register dello spazio di configurazione della funzione 0 o 1 a seconda che il blocco sia di I/O o di memoria.
Il software di inizializzazione può scoprire quanto vale b in un blocco provando a scrivere tutti 1 all'interno del BAR per poi rileggere il risultato e vedere quali bit sono stati effettivamente scritti, questo perché i bit non scrivibili oltre ai 2 o 4 meno significativi sono tutti a zero.
Il software di inizializzazione nel nostro caso è il PCI BIOS.

5.7 Come funzionano le interruzioni con questi dispositivi?

Ogni dispositivo a fino a 4 pin di interruzione **/INTA /INTB /INTC /INTD**, ogni funzione può essere collegata ad alcuni di questi piedini (anche nessuno), e questa informazione deve trovarsi nel registro di configurazione *Intr. Pin*. In particolare, se 0 a nessuno, se 1 a /INTA, 2 /INTB, e così via. Lo standard però non dice niente riguardo a quali piedini dell'APIC vanno collegati, lasciando libero spazio al progettista. Questa informazione deve essere visibile al BIOS e infatti deve essere scritta dentro il registro di configurazione *Intr. Line*.

6 DMA E BUS MASTERING

6.1 In cosa consiste il DMA?

Un dispositivo senza DMA è in grado di accedere alla memoria in 2 modi:

- Tramite interruzioni
- A controllo di programma

In entrambi i dati passano per la CPU e poi in memoria o viceversa. Con il DMA (Direct Memory Access) invece si vuole permettere al dispositivo di poter scrivere direttamente in RAM. Il software dovrà fornirgli l'indirizzo di memoria da utilizzare b e la grandezza del blocco n e poi il dispositivo, dotato di un sommatore per poter sommare man mano gli indirizzi scriverà i dati in $[b + n)$.

6.2 Come funziona in generale la scrittura con DMA?

Il dispositivo vuole scrivere in RAM e per farlo ha bisogno di scrivere i dati dentro al BUS. Nel Bus oltre la RAM è collegata anche la CPU, perciò il dispositivo ha bisogno di comunicare con la CPU per prendere controllo del BUS. Questo viene fatto con i collegamenti /HOLD e /HOLDA. In particolare:

1. Il dispositivo ha i piedini di uscita in alta impedenza
2. Quando il dispositivo vuole eseguire una operazione in RAM attiva /HOLD
3. La CPU termina l'operazione in corso, disattiva i piedini di uscita sul BUS e attiva /HOLDA
4. Il dispositivo esegue il trasferimento e disattiva /HOLD e mette i piedini di uscita in alta impedenza
5. La CPU risponde disattivando /HOLDA e riattivando i suoi piedini per continuare il normale funzionamento

In pratica il dispositivo "Ruba" cicli di BUS alla CPU, questa operazione si chiama *cycle stealing.* La CPU nel frattempo è in grado di fare operazioni che non richiedono accesso al BUS. Questa è la versione base del DMA senza contare tutto il resto dell'hardware(cache, mmu, ecc.)

6.3 Cosa comporta la presenza della cache con le operazioni di DMA?

Da una parte il fatto che ci sia la cache è un vantaggio perché la CPU può continuare a svolgere operazioni in RAM se trova il dato nella cache, però comporta tutta una serie di problematiche inerenti la validità dei dati. Le complicazioni consistono nel fatto che le operazioni di DMA potrebbero cambiare contenuti della memoria salvata in cache, rendendo così il dato in cache non consistente. A seconda della tipologia di cache si adottano soluzioni differenti.

6.4 Quali sono le problematiche e le soluzioni della scrittura DMA con la CACHE?

Le soluzioni hardware sono generalmente le stesse ma applicate in modo diverso, la tecnica dello *snooping* che consiste nel far osservare al controllore di cache quello che succede nel BUS durante una operazione in RAM da parte del dispositivo, e l'altra tecnica invece è lo *snarfing* (ingurgitare) e consiste nel far arrivare i dati in ingresso alla RAM dal dispositivo al controllore cache che può così aggiornare la cacheline

Cacheline write-though:
Nel caso di operazioni di uscita, cioè da RAM a dispositivo non ci sono problemi in quanto il dato in RAM è sicuramente coerente con quello in cache.
Il problema è quindi in ingresso, cioè dispositivo scrive in RAM.

- SOLUZIONE HARDWARE:
  Il controllore cache fa *snooping* nel BUS condiviso, se l'operazione è di lettura non fa niente mentre se l'operazione è di scrittura invalida la cacheline corrispondente. Il controllore cache può persino decidere di aggiornare direttamente la cacheline con un operazione di *snarfing* (non usata nei processori INTEL)

- SOLUZIONE SOFTWARE Per la soluzione software c'è bisogno che il processore abbia delle istruzioni in grado di comunicare con il controllore cache per far invalidare un intervallo di indirizzi. Con questa premessa basta far invalidare, nel nostro caso, gli indirizzi $[b + n)$ subito prima o subito dopo l'operazione di DMA.

Cacheline write-back e trasferimento di intere cacheline: Nel caso di cacheline *non dirty* il problema è analogo a quello delle cache write-thought e viene risolto allo stesso modo. Se il problema avviene su cacheline *dirty*.
Nel caso di transazioni di uscita (Ram to dispositivo) è sbagliato far leggere i dati direttamente dalla RAM in quanto il dato aggiornato è in cacheline.
Nel caso di transazioni in ingresso invece il problema consiste nell'invalidare la cacheline salvata in cache tenendo però conto del fatto che la cache protrebbe in caso di rimpiazamente avviare un write-back di quella cacheline durante l'operazione di DMA.

- SOLUZIONI HARDWARE:

  - Uscita in DMA (RAM to dispositivo) La soluzione hardware è lo snooping ma non basta in quanto non basta individuare che la cacheline sia *dirty* ma bisogna anche risolvere il problema, pilotare il BUS comporta una corsa con la RAM ed è da evitare. Una delle soluzioni è che lo snooping sia il dispositivo a farlo sul controllore di cache inviando anche l'indirizzo interessato e in caso di hit prendere il valore direttamente dalla cache oppure il controllore può prendere il controllo del BUS e aggiornare la RAM. Questi collegamenti possono essere fatti direttamente tra DMA e controllore oppure ottenendo il normale controllo

del BUS con /HOLD e eseguendo prima una operazione particolare ignorata dalla RAM ma nota al controllore.

  - Ingresso in DMA (dispositivo to RAM) l'operazione è lo snooping del controllore ma la cacheline deve essere invalidato senza prima essere trasferito in RAM che è il normale comportamento delle cache write-back

- SOLUZIONI SOFTWARE: La soluzione consiste nell'ordinare al controllore cache di fare write-back di un intervallo di indirizzi. Il software deve eseguire questa operazione su tutti gli indirizzi del buffer e deve farlo necessariamente PRIMA di iniziare il trasferimento. Queso funziona sia per i trasferimenti di uscita in DMA in modo che la RAM sia aggiornata prima del trasferimento che per quelle di ingresso in DMA per evitare che eventuali cacheline dirty vengano ricopiate in RAM dopo il trasferimento dei dati.

Cacheline write-back con trasferimenti generici: Il caso più complicato è quando bisogna trasferire soltanto una parte di una cacheline, ma la cacheline si trova *dirty* in cache.

- SOLUZIONE HARDWARE:

  - Uscita in DMA (RAM to dispositivo) resta immutato rispetto al caso cacheline intera, quindi con snooing del dispositivo
  - Ingresso in DMA(dispositivo to RAM) resta uguale in caso di snarfing del controllore se invece il controllore può solo invalidare senza copiare il problema deve essere risolto facendo fare snooping al dispositivo, in caso di *hit* ci sono diverse soluzioni a seconda del lavora da far fare al controllore o al dispositivo:
    * Se il controllore risponde con una write-back basta completare la normale scrittura in RAM
    * Se il controllore risponde inviando il contenuto della cacheline al dispositivo allora il dispositivo deve combinare i dati ottenuti dal controllore con quelli che deve scrivere e fare lui stesso la scrittura in RAM

    In entrambi i casi il controllore cache deve invalidare la cacheline visto che il dato corretto sta in RAM

6.5 Come funziona la memoria virtuale con i dispositivi DMA?

I dispositivi DMA non sono in nessun modo collegati all'MMU per questo usano solo indirizzi fisici. Per far funzionare correttamente i dispositivi DMA bisogna fare alcuni accorgimenti:

1. Bisogna comunicare al dispositivo non l'indirizzo virtuale b ma la sua traduzione fisica $f(b)$ altrimenti andrebbe a scrivere in indirizzi che non c'entrano nulla

2. Se l'intervallo $[f(b)+n]$ è composto da pagine non contigue allora bisogna spezzare la scrittura in parti che contengono solo frame contigui. Altrimenti il dispositivo andrebbe a scrivere in frame che non appartengono allo spazio a lui dedicato.

3. Non bisogna cambiare la traduzione degli indirizzi mentre il trasferimento è in corso.

Per quanto riguarda l'ultimo punto si pensi ad un sistema che fa *swap-in/swap-out* dei processi. Un processo P_1 inizia il trasferimento da un dispositivo alla sua memoria privata, poi il sistema decide di far spazio in RAM e fa swap-out di P_1 e al suo posto entra P_2. Il dispositivo non sa niente di questo e continuerà il trasferimento sul buffer che crede associato a P_1 ma ora ci stanno parti della memoria di P_2 creando effetti disastrosi.

6.6 Cosa sono e come funzionano i dispositivi BUS-MASTERING?

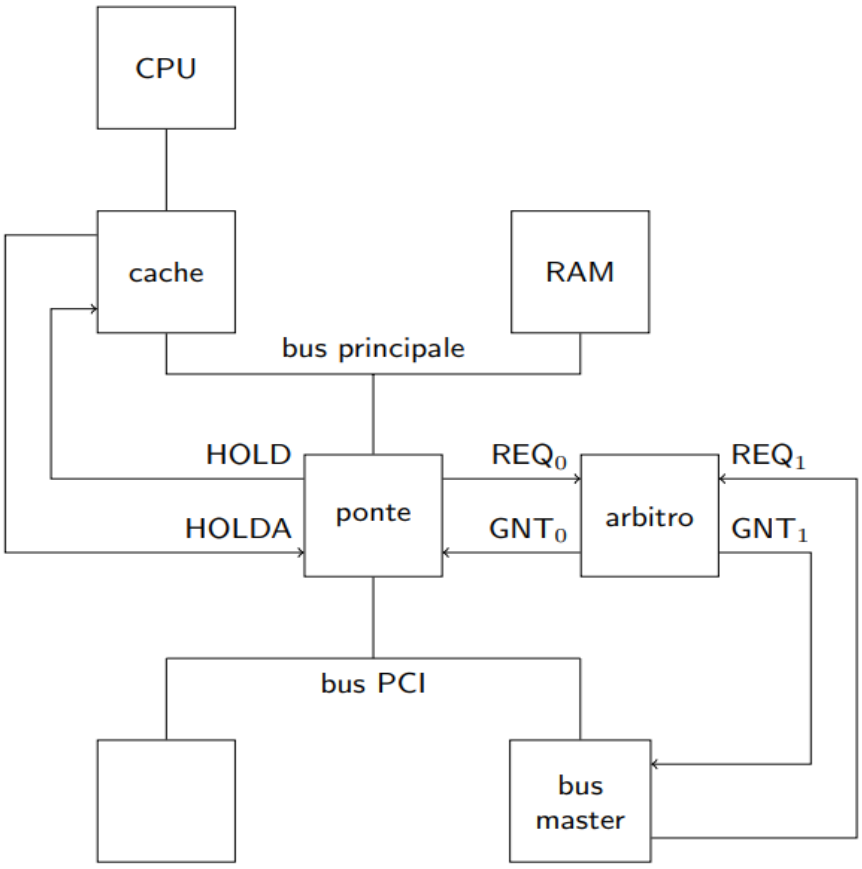


Figura 16: Architettura PCI BUS-MASTERING

I dispositivi iniziatori che possono dunque iniziare una transizione sono detti BUS-MASTER e hanno i collegamenti REQ e GNT con l'arbitro. Il Ponte è sempre BUS-MASTER in quanto attiva le transazioni sul bus-pci per conto della CPU. Nelle transazioni il Ponte è sempre l'obbiettivo. Quando un BUS-MASTER vuole iniziare una transazione con la RAM :

1. Attiva REQ e aspetta che l'arbitro mandi GNT

2. Per ottimizzare l'arbitraggio può svolgersi durante una operazione in corso, per questo prima di iniziare la transazione il dispositivo deve aspettare che una eventuale altra transazione finisce, cioè deve aspettare che /FRAME e /IRDY siano disattivati.

3. Il dispositivo accede al bus-pci

4. Il ponte è sempre l'obbiettivo e quindi risponde gli indirizzi mandati dal dispositivo

5. In caso di trasferimento da RAM a Dispositivo esegue tutte le operazioni di lettura DMA.
   In caso di trasferimento da Dispositivo a RAM il ponte salva temporaneamente il dato in un registro interno(posted writes) e in parallelo inizia le operazioni di scrittura in DMA verso la RAM.

6.7 Quali complicazioni ci sono con il meccanismo delle interruzioni con i dispositivi BUS-MASTER e quali sono le soluzioni?

La presenza del buffer sul ponte per gestire le scritture in RAM crea un problema per quanto riguarda le interruzioni, infatti quando il dispositivo finisce una transazione invia una richiesta di interruzione, ma in questo caso la richiesta potrebbe arrivare alla CPU prima che l'operazione sia effettivamente completata e il software potrebbe andare a leggere in RAM i dati che il ponte potrebbe non aver ancora caricato.

SOLUZIONE SOFTWARE:
Dato che il ponte gestisce tutte le transazioni con politica strettamente FIFO allora la routine di interruzione può cercare di leggere un registro del dispositivo prima di leggere i dati, in questo modo il ponte permetterà la lettura del registro sicuramente dopo che ha finito di trasferire tutti i dati in RAM. Le routine normalmente leggono un registro all'inizio per segnalare al dispositivo la riuscita della richiesta, quindi questa lettura basta per entrambi gli scopi.

SOLUZIONE HARDWARE:
Si crea un collegamento tra il ponte e l'APIC e fino a quando il ponte non da l'OK non viene inoltrata la richiesta alla CPU; il ponte non da l'OK fino a quando non ha finito di trasferire tutti i dati ricevuti fino a quel momento.

Un altra soluzione hardware prevede che le interruzioni vengano inoltrate sul BUS-PCI come speciali transazioni(Message Signaled Interrupts) che quindi si accodano alle normale transizioni e non ci sono problemi di corse.

7 Architettura Avanzata CPU

7.1 In cosa consiste la tecnica del pipelining?

Pipelining significa catena di montaggio. L'idea è quella di scomporre l'azione che svolge la CPU e fermarsi dopo ognuna di queste azioni: Ogni azione viene svolta da un circuito

Figura 17: Fasi di una pipeline

combinatorio diverso, mettendo un registro in mezzo a queste RC costringeremmo il processore ad aspettare 5 clock per svolgere completamente l'azione. L'idea è infatti quella di aumentare la velocità del clock. Nel mondo ideale avremmo il processore che svolge 5 richieste contemporaneamente, una in ogni fase della pipeline.

7.2 Cosa sono i processori RISC e qual'è la differenza con i processori CISC?

La tecnica della pipeline con le istruzioni dell'INTEL che conosciamo noi è impossibile, in quanto le istruzioni non sono sempre della stessa dimensione e sono complesse dal punto di vista del calcolo, si prenda ad esempio:

```
add %rax, 1000(%rbx, %rcx, 4)
```

nella fase di decodifica, per decodificare l'operando sorgente si ha bisogno di una ALU, così come nella fase di esecuzione, questo rende impossibile ad esempio avere una istruzione simile sia in fase di decodifica che di esecuzione, in quanto la ALU è una sola.
Per poter usare questa tecnica infatti bisogna che il processore sia progettato apposta per la pipeline. Questo tipo di processori si chiama RISC(*Reduction Instruction Set Computer*. Nei processori CISC(*Complex Instruction Set Computer*) come l'INTEL invece le istruzioni non sono pensate per la pipeline ma per rendere possibili alcune istruzioni e più facile la vita ai programmatori assembler, ma quelle istruzioni erano usate poco e nessuno programma in assembler.// Le istruzioni dei processori RISC sono istruzioni elementari e tutte della stessa dimensione, il formato è questo:

```
op dst, src1, src2 //Istruzioni operative
op reg, offset(base) //Istruzioni di memoria
op reg, offset //Istruzioni di salto
```

Per operare sulla memoria ci sono solo 2 istruzioni **load** e **store**.

7.3 Disegno dello schema della pipeline

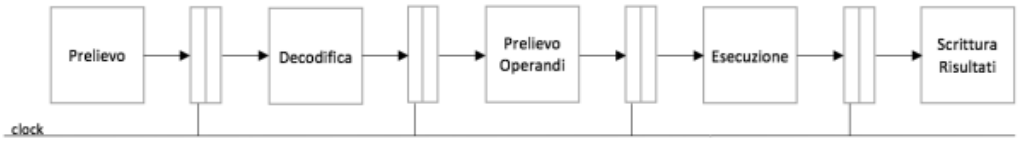


Figura 18: Schema pipeline

7.4 Quali sono i problemi legati alla pipeline?

Il problema è che ci sono alcune combinazioni di istruzioni che impediscono l'esecuzione di una istruzione ogni ciclo di clock, causando ***stalli della pipeline***.// Ci sono alee di 3 tipi:

- Alee Strutturali
- Alee sui dati
- Alee sul controllo

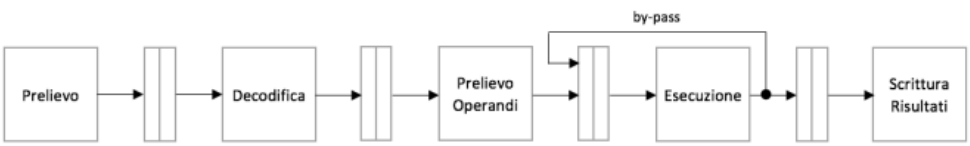
7.5 Da cosa dipendono le Alee e come si risolvono?

Le **alee strutturali** sono causate dal fatto che ci sono alcune combinazioni di istruzioni che fanno uso delle stesse risorse del processore contemporaneamente. Ci sono 2 soluzioni, una non completa e una che entra nei casi inevitabili:

- Progettare l'instruction set in modo che queste combinazioni siano ridotte al minimo.
- Si blocca il flusso della pipeline per almeno un clock fino a quando la risorsa non sia di nuovo accessibile.

Le **alee sui dati** invece sono istruzioni che usano un risultato che viene calcolato dall'istruzione precedente, ad esempio:

```
ADD R1, R2, R3
SUB R4, R1, R5
```



Quando la SUB sta in prelievo operandi in R1 non c'è il valore che mi serve in quanto la ADD lo inserirà nella fase di scrittura. Notiamo però che il dato da inserire in R1 viene

calcolato nella fase di esecuzione, quindi non mi serve aspettare che la ADD lo inserisca nel registro, questo viene implementato con un filo di bypass e la SUB prederà il risultato direttamente dalla fase di esecuzione della ADD.
Le **alee di controllo** invece si verificano nelle istruzioni di salto in cui il successivo indirizzo da prelevare non è quello successivo ma la maggior parte delle volte è uno precedente. Queste alle vengono risolte cercando di predire dove porterà l'istruzione di salto tramite predizione statica o predizione dinamica.

7.6 In cosa consistono la predizione statica e dinamica?

La predizione statica è la più semplice, e significa che una istruzione viene predetta sempre allo stesso modo, per esempio se un salto è all'indietro probabilmente è un ciclo, quindi mi aspetto di ciclare molte volte, e quindi quando ritrovo questa istruzione torno direttamente indietro.
La predizione dinamica invece è associata ad una piccola cache che serve a ricordare cosa ha fatto nel passato una istruzione, magari di una istruzione di salto, e in base al valori salvati cercano di predire il prossimo indirizzo da prelevare. In caso di predizione sbagliata significa che nelle pipeline l'istruzione di salto si trova nella fase di esecuzione e quindi le 3 istruzioni prelevate dopo sono da invalidare. Per queste istruzioni è presente un flag che se attivo quando si arriva allo stato di scrittura non conclude l'operazione.

7.7 Come mai nel primo livello di cache ci sono due cache?

Abbiamo 2 cache, una per i dati, e una per le istruzioni. Avere due cache ci serve per la pipeline perché quando uso le istruzioni che lavorano sulla memoria ad esempio la *load* vado a controllare la cache per trovare il dato, così faccio anche per il prelievo dell'istruzione, avendo una sola cache però andrei incontro ad una alea strutturale in quanto cerco di accedere alla stessa cache in 2 fasi diverse.

7.8 In cosa consiste l'esecuzione fuori ordine?

L'esecuzione fuori ordine consiste nel continuare a far eseguire istruzioni successive ad istruzioni che magari sono in bloccate perché per esempio sono delle load che hanno ricevuto un miss dalla cache e devono andare in memoria fisica a cercare il dato (centinaia di cicli di clock). Se le istruzioni successive non hanno dipendenze con quella bloccata sarebbe inutile farle aspettare. L'idea e quindi di tenere la load bloccata ed eseguire le altre.

```
for(int i=0; i<1000; i++)
    v1[i] = v2[i] + v3[i];
```

In questo codice ad esempio possiamo osservare che nessuna istruzione dipende dall'altra e quindi se magari per i = 2 il flusso si blocca le potremmo comunque continuare senza creare incongruenze.

7.9 Quali sono le dipende?

Le dipendenze sono diverse dalle aleee infatti servono per prevenirle, e sono:

- Dipendenze sui dati, una istruzione j è dipendente dai dati di una istruzione i se l'istruzione i produce un risultato che l'operazione j deve usare

  ```
  ADD R1, R2, R3
  SUB R4, R5, R1
  MUL R2, R4, R3
  ```

  La terza operazione ha una dipendenza sui dati rispetto alla seconda che ne ha uno rispetto alla prima

- Dipendenze sui nomi, istruzione i precedente a j. Ci sono 2 tipi di dipendenze:

  - Antidipendenze, l'istruzione i legge dallo stesso registro in cui legge l'istruzione j. Se faccio passare j e scrive nel registro quando poi i si sblocca leggerà un valore sbagliato.

    ```
    SUB R4, R1, R6 //legge R1
    ADD R1, R2, R3 //scrive R1
    ```

  - Dipendenza di output, entrambe le istruzioni scrivono nello stesso registro o nella stessa locazione di memoria. Se faccio passare j e scrive nel registro poi i scriverà nello stesso registro e le istruzioni dopo non leggeranno il valore aggiornato

    ```
    ADD R1, R2, R3 //scrivo R1
    SUB R1, R4, R5 //scrivo R1
    ```

- Dipendenze sul controllo, sono le istruzioni che dipendono da un salto, una istruzione j è dipendente da una istruzione di controllo i se in base al risultato della condizione di i l'istruzione j non viene processata

  ```
  Jcond fine
      i1
      i2
      i3
  fine:
  ```

  i1, i2, i3 sono dipendenti sul controllo. Il processore in realtà se c'è una istruzione di controllo tutte le istruzioni successive sono dipendenti dall'istruzione di controllo finché questa non termina

7.10 Com'è la architettura del processore per l'esecuzione fuori ordine?

L'architettura per l'esecuzione fuori ordine ha bisogno di individuare le dipendenze e cercare una soluzione quando possibile, la parte di esecuzione quindi cambia radicalmente. Per riconoscere le dipendenze uso una scoreboard in cui per ogni registro mi dice

alcune informazioni:

- bit *W*, vale 1 se in esecuzione c'è una istruzione che deve scriverci dentro
- *cont*, conta quante istruzioni voglio leggere dentro al registro
- Il contenuto del registro

in base a questo riesco a capire se ho dipendenze e che tipo di dipendenze ho sulle nuove istruzioni in arrivo, ad esempio:

```
j:  op R1, R2, R3
```

ha un dipendenza sui dati se R2 o R3 hanno W ad 1 nella scoreboard ha una dipendenza sui nomi invece se R1 ha W ad 1 o $cont \neq 0$

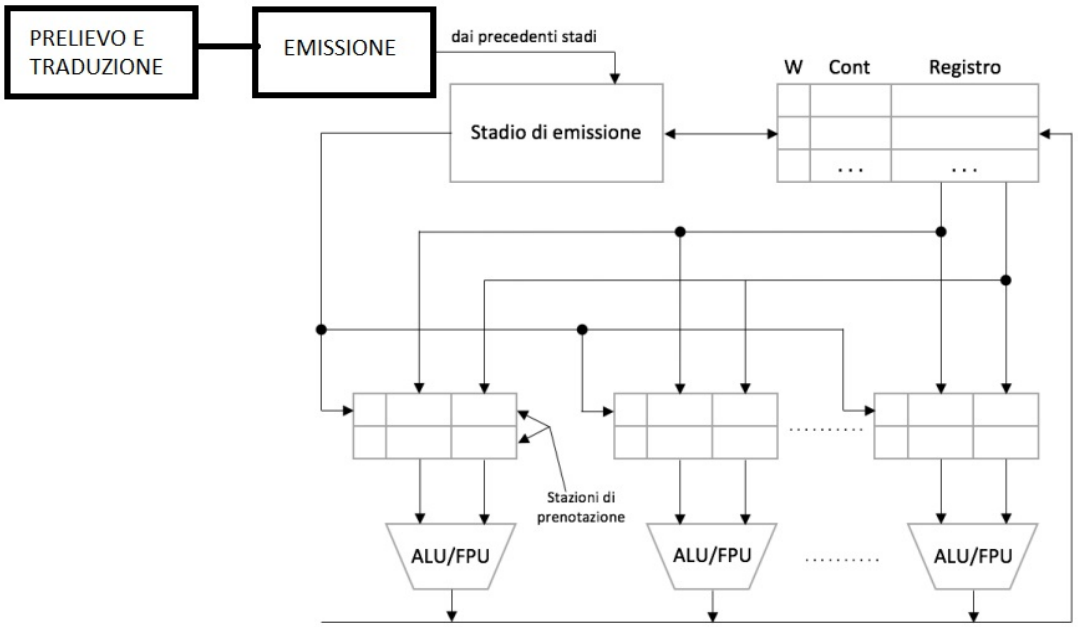


Figura 19: Esecuzione fuori ordine

Ogni circuito ALU/FPU ha una stazione di prenotazione, che corrisponde ha una coda delle istruzioni che vogliono accedervi, questo serve per limitare le alee strutturali. Se le code sono piene allora non c'è modo di risolvere la alea strutturale se non con il blocco del flusso della pipeline. Inoltre in uscita dalle ALU si ha un by-pass alla scoreboard dove in base al risultato bisogna aggiornare le informazioni sui registri.
Bisogna cercare di risolvere le dipendenze sui nomi senza ricorrere al blocco del flusso. Per esempio in questo codice c'è una dipendenza su R1:

```
DIV R2, R1, R3
ADD R1, R4, R5
SUB R6, R1, R7
```

Ci sono due modi per farli:
Algoritmo di Tomasulo, che consiste nell'utilizzare il bypass che torna sulla scoreboard. Le dipendenze sui nomi non dipendono strettamente dal registro ma dal valore che ci sta all'interno, quindi una istruzione invece che ricordarsi il registro da qui deve prendere l'informazione si ricorda da quale istruzione verrà prodotto e lo preleva direttamente da filo di by-pass, così facendo risolvo tutte le dipendenze sui nomi.
Nell'esempio la DIV sta aspettando che il valore di R1 venga prodotto da una istruzione ancora precedente ed bloccata, quindi si dimentica di R1 e aspetta il risultato nel bypass. Facendo così elimina l'antidipendenza che la ADD aveva sulla DIV, e quindi la ADD potrà scrivere tranquillamente su R1.

Un altra soluzione è quella di usare registri fisici(F1....Fn), cioè registri che non fanno parte del set di istruzioni ma che uso per salvare i dati dai registri logici (R1, R2, R3 etc.). I registri fisici sono molto di più di quelli logici in quando non essendo dentro al set di istruzioni non ho limitazioni in quanto numero. Quindi nel caso dell'esempio di prima: L'istruzione precedente alla DIV fa diventare R1 - F15 quindi:

```
DIV F2, F15, F3 //R1 ed R3 stanno in altri registri fisici
```

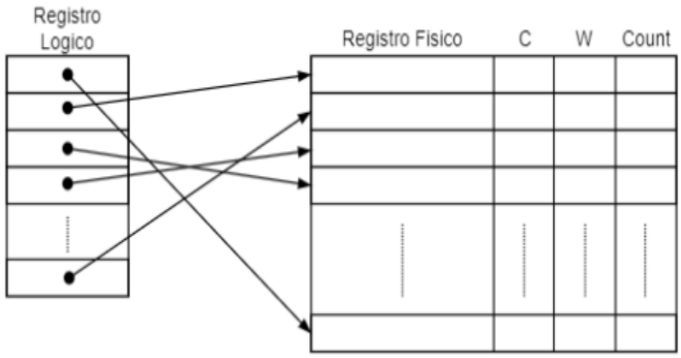
Poi la ADD invece di mettere il risultato in R1 lo mette in F17

```
DIV F2, F15, F3
ADD F17, F4, F5
```

Ora la SUB invece di leggere R1 legge F17

```
DIV F2, F15, F3
ADD F17, F4, F5
SUB F6, F17, F7
```

In pratica adesso i registri conservano solo i dati nel tempo, quindi il registro R1 cambia valore 2 volte in queste istruzioni e quindi viene salvato in 2 registri diversi.



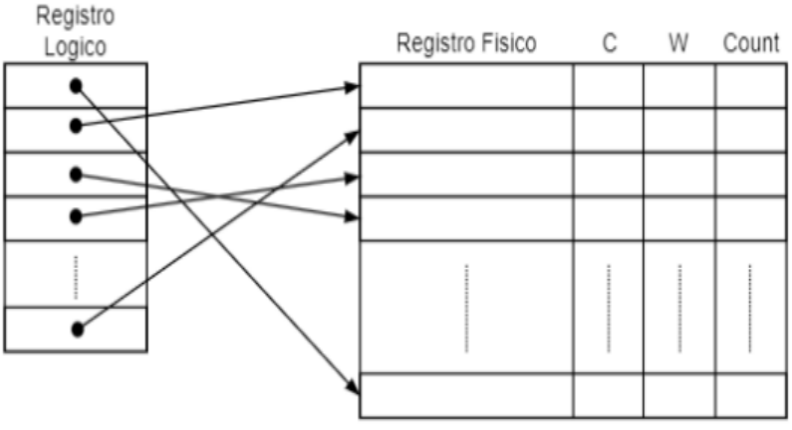
Il bit C mi dice se un registro fisico ha una corrispondenza con un regsitro logico oppure no.

Le dipendenze sul controllo invece sono più complicate, senza risolverle avremmo che ogni volta che arriva una istruzione di controllo tengo la pipeline in stallo e aspetto di sapere dove andare, se la JMP condizionale attende un dato magari dalla memoria perderei un sacco di cicli. Ci sono 2 soluzioni ottenute implementando le soluzioni sulle dipendenze dei nomi e entrambe aggiungono una nuova fase che si chiama fase di *ritiro*. Implementando l'algoritmo di Tomasulo si aggiunge una coda chiamata ROB in cui viene salvata con politica FIFO ogni istruzione, e per ogni istruzione:

- tipo della micro istruzione
- se la micro istruzione è stata completata
- in caso sia una istruzione di controllo qual'è l'esito del salto

Essendo una coda con politica FIFO l'istruzione in cima è la prima che deve uscire ma non per forza la prima ad essere completata, infatti ogni istruzione una volta completata segna nella ROB grazie ad un collegamento diretto il suo completamento. Se in cima ho una istruzione operativa, questa viene tolta dalla ROB e applicata, se l'istruzione in cima è una istruzione di salto invece, se il salto da esito negativo rispetto alla predizione del *bench predictor* allora tutte le istruzioni emesse fino a quel momento sono da annullare, quindi la ROB viene svuotata.

Se invece implemento la tecnica dei registri fisici, mi basta associare ad ogni registro logico almeno 2 registri fisici uno *speculativo* e uno *non speculativo* e fa una coda coda simile al ROB, quindi le istruzioni in coda e appena finisce quella in cima si controlla se quella sotto ha finito e così via, se invece era una istruzione di salto in base al risultato svuoto la coda oppure no. Come si fa ad annullare tutto oppure a renderlo effettivo? Dentro il registro *non speculativo* tengo il valore del registro precedente al suo ingresso



nella coda, quello *speculativo* invece lo uso per continuare l'istruzione. Se l'istruzione arriva in cima alla coda e deve completata allora copio il contenuto del registro *speculativo* nel registro *non speculativo*. Se invece l'istruzione è da invalidare a causa di una istruzione di salto con esito negativo allora inserisco il valore del registro *non speculativo* nel registro *non speculativo*.

7.11 Come si riconoscono le dipendenze nelle istruzioni di load e store?

```
/*
LOAD DST, BASE(REG)
STORE SRC, BASE(REG)
*/

LD R1, 10(R2)
ST R2, 100(R4)
```

Non posso sapere se ci sono dipendenze, in quanto non ho modo di sapere se gli indirizzi sono gli stessi. Per poter scoprire la dipendenza divido le istruzioni in una che calcola gli indirizzi e in una che sposta il contenuto, aggiungo poi dei load e store buffer al cui interno conservo gli indirizzi delle store e delle load in esecuzione e così scopro la dipendenza.
Per quanto riguarda la speculazione invece non posso completare le load e le store in quanto andrei a cambiare la memoria, salvo quindi il valore nei load e store buffer e poi quando l'istruzione deve essere completata, completo anche la scrittura.

7.12 Che succede se eseguo speculativamente una load che mi genera Page Fault?

Il Page Fault non può essere generato subito ma devo segnare nel ROB che è stato generato un Page Fault e poi quando l'istruzione esce dal ROB allora mando il lo mando.

8 ORALE

Queste domande sono le domande fatte durante l'appello 06/07/23 e l'ultima è quella che ha fatto a me l'appello del 26/07/23
Le risposte le ho scritte preparandomi all'orale, quindi prendetele con cautela perché stavo fuso.
In generale per l'orale è richiesto molto dettaglio nelle risposte e ci sono 2 regole:

1. **NON DIRE MAI NON LO SO**, meglio stai zitto. Se sta andando un po' male e dici *non lo so* considerati già al prossimo appello.

2. Devi **SEMPRE** capire chi fa cosa, se Software se Hardware e ci sono più Hardware in gioco quale di questi svolge l'azione.

8.1 In quali casi va invalidato il TLB in parte o del tutto?

Il TLB va invalidato ogni qual volta il software modifica i bit nel percorso di traduzione da indirizzo virtuale a fisico, il TLB va invalidato anche durante il cmabio di processo in quanto tutto l'albero delle traduzioni cambia. Durante una routine di *page fault* per esempio quando si leva dalla memoria fisica un frame la traduzione nel TLB va invalidata.

8.1.1 Esempio di errore causato dal non aver invalidato il TLB

Se per esempio nel software cambio dinamicamente nel percorso di traduzione i bit U/S togliendo l'autorizzazione all'utente, senza invalidare il TLB il processo poi potrebbe accedervi a livello Utente prendendo la traduzione dal vecchio valore di U/S.

8.2 Come fa l'APIC a gestire le interruzioni annidate? (Cioè le interruzioni che interrompono altre interruzioni)

Ogni interruzione ha i 4 bit più significativi del tipo che indicano la priorità, da 0 a 15. L'APIC ha 2 registri ISR e IRR che si occupano di gestire il flusso delle richieste in interruzione. Quanto una richiesta di tipo t viene inoltrata alla CPU il corrispondente bit t del registro ISR viene settato ad uno e verrà resettato all'arrivo dell'EOI. Per la gestione gestione delle richieste annidate l'APIC si comporta in questo modo, quando arriva una nuova richiesta di interruzione controlla il bit più a sinistra in ISR, che rappresenta l'interruzione con priorità più alta attualmente in esecuzione, e se la nuova richiesta ha priorità strettamente maggiore, quindi $P_r > R_s$ Allora setta ad 1 il bit t del registro ISR e inoltra la richiesta di interruzione. Quando invece arriva l'EOI l'APIC suppone che la gestione delle richieste sia di tipo LIFO e quindi l'EOI appartiene alla richiesta con priorità più alta in ISR, resetta il bit corrispondente e cerca in IRR se trova una richiesta con priorità più alta rispetto a quella con priorità più alta rimasta in ISR. Se la trova inoltra questa richiesta.

8.2.1 Come fa il processor a scrivere in EOI?

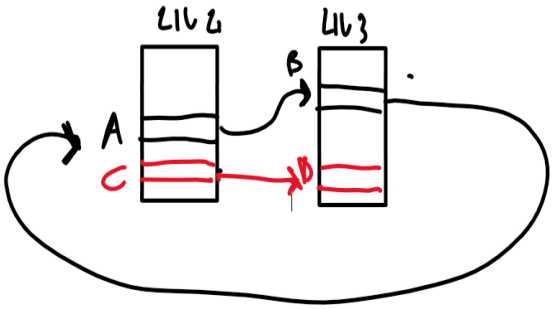
L'APIC è mappata in memoria quindi al processore basta una MOV.

8.2.2 Quali informazioni ci sono in ogni piedino dell'APIC?

Ad ogni piedino sono associati 2 registri da 32 bit e in entrambi questi registri sono conservate informazioni sul tipo di interruzione:

- Tipo dell'interruzione 8 bit
- Se l'interruzione è mascherabile o meno
- Tipo di riconoscimento, se livello o fronte
- Tipo di segnale, se attivo basso o alto

8.3 Prendo una tabella di LIV 4 e una di LIV 3, una entrata A della tabella di LIV 4 punta alla tabella di LIV 3 all'entrata B, L'entrata B punta di nuovo alla tabella di LIV 4 creando un loop. A cosa si accederebbe come indirizzi?



Facendo ABCD ottengo l'indirizzo fisico delle tabelle di LIV 2 contenuta nel descrittore di LIV 3.

8.3.1 Posso accedere alle tabelle di livello 3?

Non con questo loop AB.

8.3.2 Posso accedere alle tabelle di livello 1?

No

8.3.3 Posso accedere alle tabelle di LIV 4?

Si ripetendo il Loop quindi facendo ABAB in quanto il B contiene l'indirizzo della tabella di LIV 4

8.4 Com'è implementato il cambio di contesto? Cosa deve fare la salva_stato?

Il cambio di contesto viene implementato nel nostro sistema dalle funzioni salva stato e carica stato, cambio contesto quando ho bisogno di cambiare processo. Le mie routine di interruzione infatti iniziano sempre con una salva stato e poi con la chiamata alla routine vera e propria. Il nostro sistema ha queste strutture dati:

```
1     des_proc{
2         natl id;
3         natw livello;
4         natl precedenza;
5
6         vaddr punt_nucleo;
7
8         natl contesto[N_REG];
9         paddr cr3;
10
11        des_proc* prossimo;
12    };
13
14    des_proc* proc_table[MAX_PROC];
15    natl processi;
16
17    des_proc* esecuzione;
18    des_proc* pronti;
```

La salva stato ha il compito di salvare dentro al contesto del des_proc attualmente in esecuzione i valori dei registri del processore. Bisogna prestare particolare attenzione ai registri RIP ed RSP, il valore di RSP da salvare nel contesto deve essere quello precedente alla chiamata della salva stato. Nella salva stato noi facciamo un call e le pushq di RBX e RAX che usiamo come registri lavoro, quindi in RSP salvo il valore attuale + 24. Il valore di RSP salvato nel contesto contiene il valore di RIP salvato dal meccanismo delle interruzioni che mi permetterà dopo la carica stato di tornare al programma che ha generato l'interruzione.

8.4.1 Perché non salvo anche il registro di RIP e i FLAG?

Perché queste informazioni sono state salvate in pila sistema dall'hardware del meccanismo delle interruzioni.

8.4.2 La salva_stato deve salvare anche cr3?

No perché quando parte una routine continuo ad usare l'albero delle traduzioni del processo

8.5 Cosa deve fare la carica_stato?

La carica stato deve caricare lo stato del processo prima della chiamata della routine, quindi deve ripristinare lo stato salvato dalla salva stato. In pratica svolge il lavoro opposto e qualcosa in più. Il primo problema che viene in mente è quello di RSP, ci troviamo infatti in una CALL e quindi non posso semplicemente cambiare il valore di RSP perché poi alla fine quando parte la RET il programma salva nel punto sbagliato. All'inizio del codice viene salvato il valore del ritorno in registro tramite una POPQ, poi quando viene inserito il valore del contesto[RSP] in RSP faccio pure la PUSHQ in modo tale che la RET prenda il valore corretto. Se il processo è cambiato devo invalidare il TLB, lo scopro facilmente guardando il CR3 nel processo in esecuzione con il registro cr3, se sono diversi invalido. Sempre per la questione cambio processo bisgona cambiare il valore di punt_nucleo nel TSS, lo si fa prendendo il valore dal des_proc in esecuzione e copiandolo nel tss.

8.5.1 Quando faccio una terminate_p non posso distruggere la pila sistema in quanto mi serve poi nella carica_stato per completare le operazioni, come viene risolto?

Nel caso in cui il processo è terminato allora la carica stato si occupa di deallocare la pila sistema, chiamando la funzione distruggi_pila_precedente(); subito dopo averla cambiata mettendo il valore di RSP nel processore.

8.6 Come si crea un processo?

Un processo viene creato invocando la primitiva activate_p() che chiamerà la funzione crea_processo() che si occupa della creazione vera e propria del processo.

8.6.1 Che controlli fa la activate_p() e perché?

- Controlla che il processo che ha invocato la crea_processo non sia di livello utente, in quanto altrimenti l'utente avrebbe accesso a strutture dati protette.
- Controlla la validità dei dati, in particolare del livello, e della priorità cioè controlla che abbiano valori ammessi dal sistema e non valori a caso

8.6.2 Cosa fa la activate_p?

La activate_p oltre hai controlli chiama la crea_processo e se tutto questo è andato a buon fine inserisce il processo in lista pronti e restituisce l'id del processo appena creato

8.6.3 Cosa fa la crea_processo?

La crea processo si occupa della creazione del processo vera e propria quindi si occupa di realizzare tutte le strutture dati, prende in ingresso:

- La funzione da cui partirà il processo una volta avviato, e il suo parametro

- Il livello di privilegio
- La priorità

Non esegue controlli sui parametri perché non gli sono stati passati direttamente dall'utente.
La funzione quindi si occupa di:

- Allocare un nuovo descrittore di processo con tutte le caratteristiche richieste per questo processo, e aggiornare la proc table e il numero di processi
- Inserisce in RDI il valore del parametro per la funzione di partenza
- Allora la radice dell'albero di traduzione
- Alloca la pila sistema del processo e se il processo è a livello utente alloca anche quella utente
- Dato che il processo si trova in una routine senza mai essere passato un gate devo simulare una INT in modo che la carica stato e la IREQ successiva trovino qualcosa di consistente.

8.6.4 Come faccio a simulare una INT?

Metto dentro la pila sistema appena allocata 5 parole quadruple, prorpio quelle che il meccanismo delle interruzione avrebbe dovuto salvare. Quindi metto a partire dalla cima il RIP (l'indirizzo della funzione passata), il livello, i flags, l'indirizzo della pila, ed SS. In caso il processo sia di livello sistema metterò l'indirizzo della pila sistema

8.6.5 CAMBIO ARGOMENTO. Quale è il protocollo per la lettura e scrittura sul BUS-PCI?

Nel BUS-PCI per poter avviare una transazione in lettura o scrittura devo eseguire una sequenza di azioni:

- Devo prendere il controllo del BUS-PCI tramite un handshake con l'arbritro, in particolare il dispositivo iniziatore attiva /REQ e aspetta che l'arbitro risponda attivando /GNT. Questo però non basta per prendere il controllo in quanto l'arbitro per ottimizzare i tempi può inviare l'OK al dispositivo nonostante sul BUS ci sia una transazione in corso. Il dispositivo per procedere deve controllare che nel bus i fili /FRAME e /IRDY siano disattivati
- Inizia la fase di *indirizzamento*, Pilota i fili /AD come indirizzi e i fili /BE come command cioè comunica il tipo di operazione, se read o write e se è una operazione in memoria, io o configurazione. Attiva infine /FRAME per dare inizio alla transazione, se un obbiettivo non attiva /DEVSEL per riconoscere la transazione entro 6 cilci di clock allora si assume non ci sia nessun dispositivo.

- ISnizia la fase dei dati, e BE come bit enable poi se:
  - L'operazione è in scrittura, l'iniziatore pilota AD come dati e attiva /IRDY e attende /TRDY
  - L'operazione è in lettura, l'obbiettivo pilota AD come dati e attiva /TRDY e attende /IRDY
- Se è l'ultima istruzione allora l'iniziatore disattiva /FRAME e conclude la transazione

8.6.6 Come funziona la transazione di configurazione fisicamente?

Dal punto di vista hardware tutti i dispositivi tranne il ponte hanno un fili /IDSEL, il ponte invece è collegato a tutti questi piedini. La transazione uguale alle altre tipologie solo che il ponte attiva il piedino IDSEL corrispondente e se l'obbiettivo non attiva /DEVSEL in tempo lo standard ci dice di considerare il dispositivo assente.

8.7 A cosa serve e come funziona la CACHE?

La cache è stata introdotta per migliorare le prestazioni delle scritture e letture in memoria, in quanto la lettura in memoria RAM richiede molti cicli di clock, nei quali il processore rimane in attesa dei dati. La memoria cache invece è una memoria molto veloce e costosa, e l'idea di funzionamento è quella di salvare i dati in memoria CACHE in modo tale che il processore non debba effettuare una lettura dalla RAM ma direttamente dalla CACHE impiegando molti meno cicli di clock. Non si possono salvare tutti i dati in cache in quanto sono memorie molto piccole visto il costo, vengono salvati però i dati tenendo conto di 2 principi:

- Principio di località Spaziale, è probabile che il processore voglia accedere ad indirizzi vicini a l'ultimo a cui ha acceduto
- Principio di Località temporale, è probabile che nelle prossime istruzioni il processore riacceda all'ultimo indirizzo a cui ha acceduto.

Tutte le operazioni in cache sono gestite dal controllore CACHE che intercetta tutte le operazioni di lettura e scrittura in uscita dal processore e cerca in CACHE, se trova il dato lo passa alla CPU altrimenti svolge l'operazione di lettura in RAM al posto del processore e per il principio di Località temporale lo salva in CACHE.
La cache salva locazioni di memoria grandi 64byte dette cacheline, per riconoscere se una cacheline è presente in cache oppure no il controllore intercetta l'indirizzo generato dalla CPU e , sia 2^a il numero di cacheline massimo salvabile in cache, che è un indirizzo fisico appena tradotto dall'MMU , quindi massimo 52bit, lo divide in questo modo:

- 3 bit meno significativi di offset, 3 se le cacheline sono di 8 parole quadruple, come nel nostro casi di 64byte

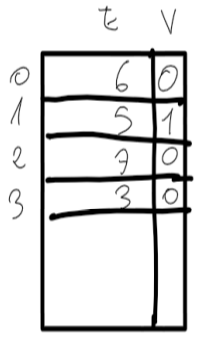
- a bit per l'indice, dato che le cacheline sono 2^a con l'indice la trovo in cache
- 49-a bit di etichetta, per rendere univoca una cacheline presa dalla RAM

La cache quindi è organizzata con due memorie diverse, una per le etichette e una per i dati. Nella memoria delle etichette viene salvata per ogni indice:

- Etichetta, serve ad indicare in quale porzione di memoria grande 2^a la cacheline è stata presa.
- V, indica se la cacheline è presente o no
- D, invece indica se la cacheline è "*dirty*" cioè se è stata acceduta in scrittura, e quindi il valore salvato in RAM non è aggiornato

I bit indice entrano nella memoria delle etichette per far cercare la cacheline all'interno, l'etichetta di quell'indice poi viene confrontata con i bit etichetta e se sono uguali significa che la cacheline è presente viene generato un bit che messo in AND con il bit di validità genera un bit di *HIT* o *MISS* se non c'è. Se la ricerca fa *hit* allora il dato nella memoria dei dati è valido. Nella memoria dei dati invece entrano in ingresso i bit enable del processore, e i bit dell'indice e dell'offset e che vengono concatenati per prelevare il dato singolo, e non la cacheline.
Il problema delle cacheline è il *conflitto fra linee* ed è dato dal fatto che le cacheline che in memoria distano 2^a cacheline hanno lo stesso indice, motivo per cui si usa l'etichetta. Questo problema nelle cache ad indirizzamento diretto è molto più accentuato.

8.7.1 Data una cache di 4 cacheline, questa memoria delle etichette e dal processore arriva: indice 1 etichetta 7, cosa succede?



Il controllore va a cerca nell'indice 1 nella memoria delle etichette (gli indici vanno da 0 ad a), trova l'indice ma l'etichetta non combacia, quindi c'è una miss. La controllore allora effettua una lettura dalla RAM, preleva il dato richiesto e lo rimpiazza in CACHE all'indice 1. Quello che succede dipende dalla politica della cache:

- Se la politica è write-though allora non serve scrive in RAM
- Se la politica è write-back è il bid D è 1 allora ricopio in RAM il valore prima di toglierlo

8.8 Siamo dentro una primitiva che deve aspettare che succeda qualcosa. Funziona?

```
bool finito;
c_primitiva(){
    while(!finito)
        schedulatore();
}
```

Non funziona

8.8.1 Perché?

Lo schedulatore estrae dalla lista pronti e inserisce nella coda esecuzione, chiamando lo schedulatore la lista esecuzione non conterrà più un solo elemento come dovrebbe ma conterrà ogni ciclo un processo diverso, fino a quando non finiscono i processi e conterrà *nullptr* generando un page fault

8.8.2 Come possiamo migliorare la cache ad indirizzamento diretto?

La cache ad indirizzamento diretto hanno il problema di conflittualità tra indici cioè non possono coesistere 2 cacheline con indice uguale. Questo problema può essere migliorati e idealmente risolto con le cache a n-vie. Le cache a n-vie sono n cache ad indirizzamento diretto messe in parallelo. I cui bit di hit o miss vengono usati come bit di comando di un multiplexer in uscita alle memorie dati (per scegliere quale far passare). I bit hit o miss vengono messi in un OR che darà il risultante bit di hit o miss della cache.

8.8.3 Se ho una CACHE di 256K a 2 vie quanti bit di indice ci saranno?

dimensione cache: 256K $= 2^{18}$
dimensione cacheline: 64byte $= 2^6$
numero di cacheline: $\frac{2^{18}}{2^6} = 2^{12} = 4K$
Ho 4K cacheline, nella cache a 2 vie ho 2K da una parte e 2K dall'altra quindi mi servono 2K $= 2^{11}$ 11 bit di indice

8.9 Schema generico di un processo esterno

```
void estern_ce(int id){
    des_ce* c = &arrayce[id];
    for(;;){
        wfi();
```

```
        }
    }
```

8.9.1 Cosa fa la wfi()?

La wfi fa essenzialmente 2 cose, manda l'EOI che comunica la fine della richiesta di interruzione, e poi si occupa di fare preemption del processo esterno chiamando lo schedulatore

8.9.2 Voglio ottimizzare la wfi e levo la salva_stato, funziona?

Nello schema base sembrerebbe di si perché tanto lo stato mi tornerebbe sempre a come lo avevo lasciato, il problema è che però il processo esterno gira ad interruzioni abilitate e quindi verrebbe interrotto e le sue strutture dati modificate.

8.10 Data una periferica con registro di ingresso di 1byte, il processo esterno deve svolgere una operazione di lettura di n byte

Un processo esterno ha il compito di leggere un nuovo byte e metterlo nel buffer

```
//struttura dati periferica
struct des_ce{
    char* buf;
    natl quanti
    natl sync;
    natl mutex;
    ioaddr iREG;
    ioaddr iSTS;
};


void estern_ce(int id){
    des_ce* t = &arrayce[id];
    for(;;){
        t->quanti--;
        if(t->quanti == 0){
            output(0, t->iSTS);
            sem_sygnal(t->sync);
        }
        char c = inputb(t->iREG);
        *buf = c;
        *buf++;
        wfi();
    }
}
```

Recupero il descrittore, decremento quanti, se quanti dopo il decremento è 0 allora finisco la transizione scrivendo nel registro di stato e metto il gettone nel semaforo.

8.10.1 Come funziona la funzione inputb?

È una funzione scritta in assembly che fa una istruzione *in*

8.10.2 Supponiamo di esserci dimenticati di disabilitare le interruzioni, quando la periferica può mandarmi una nuova interruzione?

Teoricamente subito dopo la lettura del registro iREG.

8.10.3 Quando però viene inoltrata la richiesta?

Dopo la wfi() che manda l'EOI

8.10.4 Quando il processore la vede?

La vede dopo la IRETQ della wfi in quanto carica nel processore il registro dei flag con IF e quindi gira ad interruzioni abilitate.

8.10.5 Quel buffer è utente o sistema?

Può essere entrambi dipende dal processo, se il processo che ha chiamato la primitiva è un processo a livello utente allora non è accettabile che il buffer sia a livello sistema, se invece il processo che ha chiamato la primitiva è a livello sistema allora è accettabile che stia nella parte sistema

8.11 DMA, che cosa è, perché si usa, svantaggi e vantaggi e problemi

DMA sta per Direct Memory Access ed è un tipo di trasferimento dati. Consiste nel permettere al dispositivo di scrivere direttamente nella memoria senza prima inviare i dati alla CPU, i vantaggi stanno nel fatto che il processore può lavorare in parallelo mentre la periferica scrive in RAM. I problemi nascono su più fronti, sicuramente uno è con la cache:
Se la periferica legge un dato in RAM e la cache non è write-though allora non è detto che quel risultato sia valido, oppure se la periferica scrive in RAM ma il dato è presente in cache il processore non lo vede. Questi problemi sono risolti da 2 tecniche, lo snooping da parte del controllore o del dispositivo, e lo snarfing da parte del controllore. Con lo snooping il controllore di cache vede sul BUS che tipo di operazione il dispositivo sta effettuando e, se l'operazione è di lettura e la cacheline è *dirty* allora intercetta la lettura e fa write back del dato oppure gli passa direttamente il dato. Se l'operazione è in scrittura il controllore può fare snarfing cioè aggiorna la cache oppure invalida la cacheline corrispondente.

8.11.1 Come facciamo a configurare una operazione in DMA?

(NON SO SE GIUSTO) Con una primitiva. Essendo una periferica che adopera in DMA bisogna comunicare negli appositi registri dove quanti dati dovrà salvare e dove, in quanto la periferica opera solo con indirizzi fisici:

```
1   //struttura dati periferica
2   struct des_ce{
3       char* buf;
4       natl quanti
5       natl sync;
6       natl mutex;
7       ioaddr iREG;
8       ioaddr iSTS;
9   };
10
11  void ce_dmaread(natl id, char* buf, natl quanti){
12      \\controllo id
13      \\controllo se posso accedere a buffer
14      des_ce* p = &array_ce[id];
15      sem_wait(p->mutex);
16      paddr t = trasforma(buf);
17      outputl(t, p->iBUF);
18      outputl(quanti, p->iLEN);
19      outputl(1, p->CMD);
20      sem_wait(p->sync);
21      sem_signal(p->mutex);
22  }
23
```

Non essendo una primitiva si gira ad interruzioni abilitate quindi ho bisogno di far si che nessun altro processo interferisca e lo faccio mettendo un semaforo così chiunque voglia usare la periferica si dovrà mettere in coda.

8.12 Proviamo ad invertire CACHE ed MMU

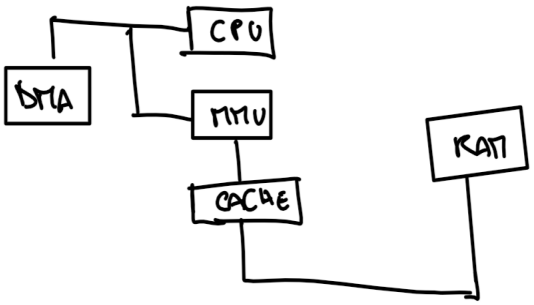
(Domanda per la lode) Un problema è quello che ci possono essere più indirizzi virtuali in un indirizzo fisico, l'unica cosa che hanno in comune è l'offset. Se faccio l'indice in modo tale che sia parte dell'offset allora sono sicuro che nelle cache ad indirizzamento diretto verranno viste come stessa cacheline. Il problema a questo punto è che la cache sarebbe molto piccola in quanto l'indice è più piccolo dell'offset, per ovviare a questo posso usare cache a più vie e usando come etichetta l'indirizzo fisico tradotto.

8.13 Che problemi presta la scrittura in DMA con la MMU?

I dispositivi in DMA sono dispositivi che operano con indirizzi fisici in quanto non possono aver accesso alle traduzioni, per questo bisogna fare alcuni accorgimenti:

- Bisogna comunicare al dispositivo non l'indirizzo virtuale ma l'indirizzo fisico
- Se l'indirizzo virtuale attraversa più pagine tradotte in frame non consecutivi allora devo spezzare la scrittura in parti che contengono solo frame contigui
- Non bisogna cambiare la traduzione degli indirizzi mentre il dispositivo è in trasferimento, cioè bisogna evitare di fare swap-out della memoria del processo durante il trasferimento.

8.13.1 Potrei risolvere il problema mettendo il DMA di fianco alla MMU?



Non funziona perché in un sistema di multiprocesso la radice dell'albero di traduzione punta all'albero del processo in esecuzione, quindi il DMA mentre scrive e passa per l'MMU e avviene un cambio di processo gli indrizzi non sarebbero più validi.

8.13.2 È risolvibile? O questa architettura è da buttare?

Può essere risolvibile se il dispositivo scrive sempre nelle parti condivise dei processi e questi usino sempre le stesse traduzioni.

8.14 Qual'è la struttura del processore che abbiamo usato che ci permette di eseguire l'esecuzione speculativa?

Per poter implementare l'esecuzione speculativa i processori moderni devono essere processori RISC che usano la pipeline, nella esecuzione normale con la pipeline le fase del processore sono divise in 5: prelievo, decodifica, esecuzione divisa in: prelievo, esecuzione, scrittura.
Nell'esecuzione speculativa invece tutta la parte di esecuzione non è più procedurale. L'idea dell'esecuzione speculativa è quella di evitare che la pipeline vada in stallo, ad esempio:
Ho una istruzione che richiede una lettura in memoria, la cache fa miss e devo leggere in RAM. Non posso pretendere che il processore perda 200 e più cicli di clock per aspettare. L'idea è quindi quella di iniziare ad eseguire tutte le istruzioni che non hanno bisogno di quel dato da leggere in modo che mentre si esegue la transazione in memoria la CPU

continui il flusso di esecuzione. Prendiamo 2 istruzioni i e j tali per cui i viene prima di j e vediamo quali sono i problemi:

- Alee strutturali, avvengono quando 2 istruzioni vogliono accedere alla stessa risorse, ad esempio una ALU o la CACHE. Un esempio di alea strutturale è quando 2 istruzioni vogliono accedere alla ALU contemporaneamente, una delle due deve mettersi in coda.
- Alee sui dati, avvengono quando una istruzione j deve usare un dato prodotto da una istruzione i precedente che però non lo ha ancora prodotto
- Alee di controllo, avvengono durante le istruzioni di salto, quando l'indirizzo da prelevare non è il successivo ma la maggior parte delle volte è un indirizzo precedente

Tutte queste alee possono essere risolte con degli stalli i modo poco efficiente. Quindi per progettare un sistema ad esecuzione speculativa devo tener conto delle dipende tra istruzioni, presa una istruzione i precedente alla istruzione j:

- dipendenze sui dati, se una istruzione i produce un risultato e l'istruzione j legge quel risultato allora si dice che j ha una dipendeza sui dati da i
- dipendenze sui nomi, sono di 2 tipi:
  1. Antidipendenza, L'istruzione i legge da un registro che l'istruzione j scrive, quindi se j va in esecuzione prima di i allora i legge un valore sbagliato
  2. Dipendenza di output, i e j scrivono nello stesso registro se però j viene eseguita prima e i dopo allora le istruzioni successive leggeranno un dato sbagliato
- Dipendenze sul controllo, se una istruzione i è una istruzione di controllo allora tutte le istruzioni successive hanno dipendenza sul controllo da i

Le dipendenze sui dati non possono essere risolte, le dipendenze sui nomi e sul controllo si. Per risolvere le dipendenze sui nomi ci sono 2 modi:
Algoritmo di Tomasulo, consiste nel far ricordare alle istruzioni non più il registro dal quale devono leggere ma l'istruzione che produrrà il dato che gli serve. In questo modo le antidipendenza sono risolte in quanto ora l'istruzione j può scrivere nel registro senza che i legga un risultato sbagliato.
Registri fisici, i registri fisici sono registri che usa solo il processore, non sono utilizzabili dalle istruzioni, quindi non hanno un limite numerico. L'idea è quella di far si che ogni registro logico venga associato ad uno più registri fisici in cui viene inserito un valore, ad esempio:

```
1    DIV R1, R2, R3
2    ADD R2, R4, R5
3    SUB R6, R2, R7
```

Lo trasformiamo usando gli indirizzi fisici

```
DIV F1, F15, F3
ADD F17, F4, F5
SUB F6, F17, F7
```

In questo modo la antidipendenza su R2 viene risolta.
Le dipendenze di controllo invece posso essere anche loro risolte in due modi entrambe ottenute aggiungendo una nuova fase, detta fase di ritiro:

Si aggiunge all'algoritmo di Tomasulo una coda di istruzioni in uscita, tutte le istruzioni possono essere completate ma vengono terminate solo passando per questa coda dove l'uscita è strettamente FIFO quindi se l'istruzione in cima non è terminata allora le altre istruzioni in coda anche se completate aspettano. Così facendo quando arriva una istruzione di controllo tramite il bench predictor le altre istruzioni presunte successive possono essere emesse e completate ma tanto non verranno terminate fino a quando l'istruzione di controllo non viene completata, una volta completata l'istruzione di controllo scopro se il bench predictor ha indovinato il flusso e continuare altrimenti svuoto la coda ed è come se si fosse andato in stallo la pipeline.

Con la tecnica dei registri invece associo ad ogni registro logico 2 registri fisici, uno speculativo e uno non speculativo e nel primo conservo il valore prima dell'istruzione di salto mentre l'altro lo si usa per le presunte istruzioni dopo il salto. Uso la stessa tecnica della coda solo che quando scopro se il bench predictor ha indovinato sposto il valore del registro speculativo in quello non speculativo se invece non ha indovinato sposto quello speculativo in quello non speculativo.

8.14.1 Esempio di alea sui dati

```
ADD R1, R2, R3
SUB R4, R1, R5
```

Se la SUB sta in fase di prelievo operandi e la ADD in fase di esecuzione allora la SUB non troverebbe in R1 il dato che gli serve come sorgente, per ovviare a questo tra la fase di esecuzione e quella di prelievo operandi si inserisce un bypass in modo tale che la SUB non debba aspettare che il risultato venga caricato ma può prenderlo direttamente appena il dato è pronto.

8.15 Prendo un processo esterno e tolgo la EOI dalla wfi() e la metto prima, posso farlo, funziona?

```
void estern_ce(int id){
    des_ce* c = &array_ce[id];
    for(;;){
        apic_send_eoi();
        wfi_no_eoi();
```

```
6           }
7       }
```

Si posso farlo perché il processo esterno sta a livello sistema e quindi può richiamare la funzione.
Dopo che mando l'EOI potrei ricevere una richiesta di interruzione esterna di nuovo dalla periferica. Supponiamo quindi che arrivi una richiesta di interruzione tra la EOI e la WFI. Il flusso di esecuzione si ferma e si attraversa un gate, si salvano i valori e il RIP che viene salvato punta alla wfi_no_eoi(). Salva lo stato del processo e inserisce il processo in coda pronti e poi il nuovo processo esterno in esecuzione. Il problema ora è questo l'interruzione arrivata era della stesso dispositivo quindi l'handler ha messo in esecuzione lo stesso processo esterno. La situazione sembra sbagliata in quanto ora ho lo stesso processo sia in esecuzione che in coda pronti. Quindi ho 2 processi in giro **ma un solo descrittore di processo**. L'handler continua e mi carica lo stato nel descrittore e poi fa la iretq, mettendo in RIP l'indirizzo della wfi_no_eoi() salvata prima. Quindi il processo esterno chiama la wfi_no_eoi() e salva in pila RIP che punta dopo la wfi, quindi all'inizio del ciclo for, e chiama schedulatore che metterà in esecuzione lo stesso processo esterno che però era rimasto in lista pronti. Fa la carica stato e la iretq e riparte il ciclo. Quindi passando per stati inconsistenti ma alla fine siamo finiti in uno stato consistente. In pratica all'inizio sembrava di aver perso una richiesta di interruzione visto che il processo è finito in coda pronti, però la nuova richiesta di interruzione mi ha terminato quella precedente e la quella precedente ha terminato l'altra.

8.16 Traduzione da indirizzo virtuale a fisico, come funziona? chi la fa? Perché si fa?

8.17 Supponiamo di voler scrivere un programma che usi interruzioni in una macchina vuota

Il problema principale di avere una macchina vuota con il meccanismo delle interruzioni è il non avere una IDT, la prima cosa che dovrei fare quindi è mappare in memoria la tabella IDT e inserire l'indirizzo fisico della tabella nel registro IDTR tramite l'istruzione lidt. Poi dovrei scrivere una routine per la gestione dell'eccezione e salvare l'indirizzo nel gate associato. Poi mi manca una pila sistema, in cui con il meccanismo delle interruzioni avrei salvato le 5 parole quadruple. L'indirizzo alla pila va salvato nel campo punt_nucleo del TSS corrente. Ma io non ho il TSS corrente ne la GDT, quindi costruisco anche questi e inserisco in gdtr l'indirizzo della GDT e in tr quello del ts relativo alla GDT. Non posso scrivere direttamente la routine di interruzione in c++ perché ho bisogno di svolgere il compito inverso delle interruzioni, cioè riprendere le 5 parole lunghe salvate in pila e messe dentro il processore, per farlo devo passare prima dall'assembler, chiamare la routine in c++ e poi fare la iretq.

8.18 Come fa il debugger a realizzare le funzionalità step-by-step e breakpoint?

Per realizzare la modalità single-step il processore ha un flag TF che se settato genera una interruzione dopo ogni istruzione. In particolare il processore prima di eseguire una istruzione controlla se il flag TF è settato, in caso lo sia dopo l'istruzione genera l'eccezione di tipo1. Le funzioni enable_single_step e disable_single_step si occupano rispettivamente di abilitare e disabilitare il flag TF. Per il breakpoint invece l'eccezione da generare e il tipo 3 (0xCC). Per generarla metto al posto dell'indirizzo dell'istruzione in cui voglio mettere il breakpoint OxCC occupandomi di salvare in una struttura dati globale l'indirizzo corretto, e quando quella istruzione verrà chiamata partirà invece l'eccezione e la routine a_debug e c_debug che bloccano l'esecuzione. Queste routine si occupano anche di rimettere il valore dell'istruzione che avevo salvato e riprendere la normale esecuzione

8.19 Perché usiamo il meccanismo della protezione?

Il meccanismo della protezione consiste nel non permettere all'utente di svolgere determinate attività nel nostro sistema. Il meccanismo della protezione consiste quindi nell'avere dei livelli di privilegio, utente e sistema, e avere determinate zone della memoria e istruzioni non utilizzabili a livello sistema in modo da non compromettere l'integrità dello stesso. Si vuole permettere all'utente di accedere in modo controllato comunque ad alcune funzionalità del modulo sistema e lo si può fare facendo attraversare un gate della idt. Infatti il livello di privilegio può essere innalzato solo attraversano un gate e abbassato solo tramite la IRETQ. Per implementare questo si è resa privilegiata l'istruzione lidtr che serve per modificare il contenuto della tabella IDT (mappata nel modulo sistema) e introducendo le interruzioni via software tramite l'istruzione **INT$tipo** che permette all'utente di attraversare un gate e raggiungere una routine. Altre istruzioni vietate all'utente sono: **lgdt** per caricare l'indirizzo base della GDT, la in e la out, e la hlt.

8.20 Cosa contengono i gate della IDT e cosa fa il processore quando attraversa un gate?

Ogni gate della IDT è grande 16byte, di cui 8 contengono l'indirizzo alla routine di interruzione. Nei restanti byte ho informazioni per quanto riguarda il gate:

- Bit P (present), indica se un gate è presente o no
- campo DPL (descriptor privilage level), indica il livello minimo che devo avere per accedere al gate
- campo L(level), indica il livello che dovrò avere subito dopo aver attraversato il gate
- I/T (interrupt/trap), indica se l'interruzione è di tipo interrupt o trap

Quando si attraversa un gate della IDT l'obbiettivo è andare alla routine di interruzione, ma prima vanno fatte alcune cose:

- Se P=0 allora il gate non è presente e genera una eccezione
- Se il contenuto di cs nel processore è minore di DPL allora si genera una interruzione di protezione
- Se il contenuto di cs è maggiore di L allora eccezione di protezione in quanto l'attraversamento di un gate non può abbassare il livello di privilegio
- Salva in SRSR il contenuto di RSP in caso di cambio pila
- Salva nella pila sistema del processo 5 parole quadruple, in ordine dalla cima verso al basso RIP, cs, rflags, srsp, ss
- Se L è diverso da cs esegue un cambio pila, dato che se è diverso significa che da utente si passa a sistema trova il valore della pila sistema dentro al TSS corrente nel campo punt_nucleo
- Se I/T è interrupt allora setta il flag IF
- resetta TF per disattivare il single step
- salta alla routine

8.21 Semafori, come si implementano e a caso servono?

Il meccanismo dei semafori è stato introdotto per permettere ad un sistema multiprocesso di avere delle parti condivise tra processi e per cui bisogna accedere con un determinato ordine per non rendere inconsistenti i dati. L'idea è quella di avere una struttura dati che permetta di risolvere i problemi della sincronizzazione e della mutua esclusione, per farlo si è pensato di avere un descrittore di semafori al quale il processore deve accedere prima di avere accesso al codice che si vuole proteggere, e può farlo solo se all'interno del descrittore trova dei gettoni, altrimenti si mette in coda. Nel nucleo sono implementi da un descrittore di semafori e da un array di descrittori più tutte le primitive associate:

```
1    struct des_sem{
2        int counter;
3        des_proc* p;
4    };
```

Il descrittore di semaforo è costituito da un intero counter che serve per contare i gettoni e una lista di processi in coda che aspettano che un gettone venga inserito nel semaforo. Per inserire ed estrarre gettoni dal semaforo il sistema ha messo a disposizione 2 primitive semaforiche, la sem_wait e la sem_signal che si occupano rispettivamente di togliere un gettone dal descrittore e mettere un gettone nel descrittore. Entrambe si occupano anche di gestire la coda, infatti se un processo invoca la sem_wait su un semaforo ma

se non ci sono gettoni la primitiva inserisce il processo nella coda del semaforo e chiama lo schedulatore. La sem_signal invece se aggiungendo il gettone il counter del semaforo diventa positivo allora fa preemption del processo in esecuzione, inserisce il primo processo bloccato in lista pronti e chiama lo schedulatore. Un altra primitiva e la sem_ini che si occupa di inizializzare un semaforo dato in ingresso il numero di gettoni iniziali. Il numero totale di semafori possibili viene diviso all'interno dell'array di descrittori in una parte per l'utente e una per il sistema.
Combinando queste primitive risolvo i problemi della mutua esclusione e della sincronizzazione:

- Mutua Esclusione, alloco un semaforo con all'interno un solo gettone in modo da far lavorare solo un processo per volta:

```
1     natl mutex = sem_ini(1);
2     ...
3     ...
4     sem_wait(mutex);
5     \\ azioni da svolgere in mutua esclusione
6     sem_signal(mutex);
```

- Sincronizzazione, siano 2 processi P_a e P_b che svolgono rispettivamente le azioni A e B, voglio che B venga svolta sempre dopo A

```
1         natl sync = sem_ini(0);
2
3 Pa:     A;
4         sem_signalsync);
5         ...
6         ...
7         ...
8 Pb:     sem_wait(sync);
9         B;
```

  In questo modo se P_a non ha svolto A ma P_b cerca di svolgere B, nel semaforo non ci sono gettoni e rimane bloccato in attesa che P_a invochi la sem_wait per inserire un gettone. Di contro se P_a svolge A e inserisce un gettone allora P_b non rimane bloccato.

8.22 Com'è fatta la struttura dati del timer nel nucleo?

da rispondere ()

8.23 DMA e CACHE

I dispositivi DMA sono dispositivi che scrivono o leggono direttamente dalla RAM senza passare per il processore, questo comporta problemi per quanto riguarda la CACHE

infatti il controllore cache se i dati che sta cercando il processore sono in cache gli passa quelli o modifica direttamente quelli. Il DMA d'altro canto è in grado di scrivere e leggere in RAM quindi il processore senza le giuste interazioni tra questi 2 dispositivi rischia usare dati sbagliati così come il dispositivo DMA. Bisogna quindi che i due dispositivi comunichino in qualche modo. A seconda del tipo di transazione e del tipo di cache queste interazioni cambiano.
Cache write though con trasferimento di intere cacheline: Il problema sta in uscita in DMA in quanto il dato in cache deve essere invalidato o aggiornato:

- SOLUZIONI HARDWARE: il controllore cache fa *snooping* nel bus e capisce che si sta effettuando una scrittura in RAM, quindi invalida la cacheline corrispondente o fa *snarfing* e la aggiorna.

- SOLUZIONI SOFTWARE: per risolvere via software bisognerebbe avere una istruzione in grado di dire alla cache di invalidare le cacheline dato un indirizzo

Cache write back con trasferimento di intere cacheline:

- SOLUZIONI HARDWARE: in caso di uscita in DMA, se la cacheline è dirty il dato in cache non è valido, si risolve facendo fare snooping al dispositivo sulla cache è in caso di cacheline dirty il controllore cache può passare il dato direttamente al dispositivo o fare write back, in entrambi i casi il controllore prende controllo del bus.
  in caso di ingresso in DMA allora il controllore fa snooping e in caso di cacheline dirty invalida la cacheline senza fare write back

- SOLUZIONI SOFTWARE: La soluzione è ordinare al controllore cache di fare write back di un dato intervallo di indirizzi necessariamente PRIMA di fare il trasferimento.

Cache write back con trasferimenti generici

- SOLUZIONI HARDWARE: Il controllore cache fa snooping e se è in grado di fare snarfing allora combina i dati se invece può solo invalidare la cacheline fa snooping il dispositivo e in caso di hit se il controllore risponde con una write back si completa la scrittura in ram se invece il controllore risponde inviando la cacheline allora il dispositivo combina i dati e scrive in RAM.

8.24 Come è implementata la coda del timer?

Da rispondere

8.24.1 Cosa fa la routine del timer? (delay)

Da rispondere

8.24.2 Nella coda con i contatori già decrementati quindi, cosa si orrimizza e cosa si rallenta?

Ottimizzo la routine del Timer, peggiorando la delay, in quanto il timer deve solo decrementare il primo elemento della lista, mentre la delay deve inserire in un lista ordinata e poi sistemare tutti i contatori.

8.25 DMA con PCI, come si svolge il trasferimento? Disegno e descrizione dettagliata delle transazioni Bus_Master alla RAM

Da rispondere

8.26 Non faccio la finestra nella memoria fisica, quindi non ho nulla di mappato, cosa faccio se arriva una interruzione?

Da rispondere, la domanda consiste nel farti ragione sul meccanismo delle interruzioni e facendoti dire cosa e mappato dove. Per esempio: il processore cerca la IDT ma non è mappata, allora page fault. Poi, mappiamo la IDT e continuiamo ...

8.27 Come potrebbe un processo esterno mappato nella parte utente mandare l'EOI?

Non potrebbe in quando l'EOI è mappato in memoria sistema, si può però mappare nella memoria utente del processo una traduzioni con i permessi utente per poter scrivere nell'EOI.

8.27.1 Quali altri problemi troverebbe il processo esterno a lavorare con privilegio utente?

Non potrebbe accedere ai registri dei dispositivi in quanto sono mappati nella zona di memoria riservata all'IO

8.27.2 Che privilegio ci vuole per accedere alla memoria I/O?

Ci vuole l'IO Privilage e bisogna avere il flag IOPL settato.

8.27.3 Cosa comporta avere IOPL settato nei flags?

Permette di poter fare la *in* e la *out* e di usare le istruzioni *cli* e *sti*.

8.27.4 Quindi è possibile avere un processo esterno a livello utente?

No perché non vogliamo che un processo utente sia in grado di usare cli e sti per disabilitare le interruzioni.