

DOMANDE SONZA

Si descrivano brevemente i 3 conflitti (hazard) che possono mandare in stallo una pipeline

Structural hazard : causati da conflitto di risorse (utilizzo di unità funzionali comuni)

Data hazard : un'istruzione necessita di un dato elaborato dall'istruzione che la precede (RAW WAR WAW)

Control hazard : dipendono dai salti *condizionati* nella pipeline.

Si descriva il meccanismo pseudo-LRU di sostituzione delle linee di cache utilizzato nel 486

Ad ogni insieme (composto da 4 linee L0, L1, L2 e L3) sono associati 3 bit B0, B1 e B2. Ogni volta che si fa accesso ad un insieme, i 3 bit vengono aggiornati secondo le seguenti regole:

- accesso ad L0 o L1: $B0 \leftarrow 1$
- accesso ad L0: $B1 \leftarrow 1$
- accesso ad L1: $B1 \leftarrow 0$
- accesso ad L2 o L3: $B0 \leftarrow 0$
- accesso ad L2: $B2 \leftarrow 1$
- accesso ad L3: $B2 \leftarrow 0$

All'atto di caricare un blocco in memoria, la scelta su dove metterlo nella cache viene fatta in base al seguente albero di decisione:

- se c'è una linea nell'insieme che non è valida, il nuovo blocco viene inserito al suo posto; altrimenti
- si testa B0: se ne deduce se l'ultima linea utilizzata è stata L0 o L1, oppure L2 o L3
- si testa B1:
 - se $B0=0$ e $B1=0$ si rimpiazza L0
 - se $B0=0$ e $B1=1$ si rimpiazza L1
 - se $B0=1$ e $B2=0$ si rimpiazza L2
 - se $B0=1$ e $B2=1$ si rimpiazza L3.

Si descriva brevemente l'organizzazione ed il funzionamento dell'architettura di Tomasulo.

L'architettura di Tomasulo è alla base di procesori superscalari. A monte dell'istruzione unit c'è l'unità di fetch. A valle della Instruction queue c'è un'unità chiamata ISSUE che decodifica le istruzioni prelevate dall'Instruction queue e le assegna alle unità funzionali. La novità sono le reservation station che compaiono a monte delle unità funzionali FP:

Tomasulo, pensando al problema degli hazard di dato e strutturali, pensò di accodare alle reservation station (buffer in cui sono accodate le istruzioni che competono ad una unità funzionale) le istruzioni che vanno eseguite da una

particolare unità funzionale. Ogni unità funzionale ha una reservation station. L'ISSUE verifica se c'è spazio in una reservation station di destinazione e se non c'è spazio si verifica un hazard strutturale e quell'istruzione viene memorizzata in un buffer interno della ISSUE stessa. Per tutte le istruzioni usuali (ADD, SUB,...) la ISSUE assegna una istruzione alla relativa reservation station, dopodiché va a vedere di quali operandi ha bisogno. Dentro il register file, associato ad ogni registro c'è un flag che mi dice se quel registro ha un valore stabile o sta per cambiare: se il valore sta per cambiare dovrò attendere, fino a quando l'istruzione non avrà modificato il dato e resetto il flag. Se gli operandi sono stabili li guardo e li metto nella reservation station. Se uno o entrambi gli operandi non sono disponibili nella reservation station setterò dei flag per dire che quell'istruzione non può partire finché i registri che richiede come operandi non conterranno un valore stabile, il che richiede il completamento delle istruzioni che modificano i registri richiesti. Il Common Data Bus CDB permette la notifica su quali operandi sono disponibili: qui la reservation station controllerà se gli operandi competono ad una istruzione che li aspettava, e mentre questi passano sul CDB per essere scritti sul register file lei li preleva per fornirli all'istruzione. L'istruzione, che fino ad allora era bloccata, riceverà gli operandi e potrà proseguire. Sono quindi le reservation station a tenere memoria degli operandi che attendono e a sapere da quale unità funzionale l'attendono. Quando sul common data bus le reservation station si preleva il dato o i dati e li passa all'istruzione che li richiedeva. Ovviamente ci sarà un'unità funzionale che scrive il risultato nel CDB, e poi anche se sono 2 o 3 le reservation station ad attendere ogniuna di loro prenderà il dato e lo fornirà all'istruzione che lo attende. Nota: I dati sono etichettati sul CDB: e questo che garantisce che la reservation station possa sapere se il dato gli interessa o meno. C'è da dire anche che alle reservation stations viene comunicato qual è il dato che gli interessa (uno o più di uno) e anche qual è eventualmente l'unità funzionale che lo scriverà sul CDB.

Si illustri brevemente l'architettura ed il funzionamento di un processore VLIW, sottolineando vantaggi e svantaggi rispetto ai processori superscalari ed indicandone le principali aree di applicazione.

L'idea che sta alla base dei processori VLIW è quella di aumentare il numero di unità funzionali, e si può arrivare tranquillamente anche a 10 unità funzionali molte delle quali tutte uguali tra loro. Dopodiché queste unità lavoreranno tutte in parallelo. Il nome deriva dal fatto che il compilatore deve trovare a questo punto dei pacchetti di istruzioni che permettano di sfruttare al max tutte le unità funzionali. Le istruzioni di questo tipo di processori sono molto grandi ed inglobano tutte quelle istruzioni che devono rientrare nel pacchetto, e vengono inviate alle unità funzionali che le lavoreranno tutte in parallelo. Il tipico VLIW ha un certo numero di unità funzionali, più o meno uguali fra loro.

Si può vedere l'istruzione di un VLIW come la giusta opposizione di istruzioni di un RISC. Ovviamente questo complica il lavoro del compilatore. E' scontato che ad ogni colpo di clock non è detto che trovi un numero di istruzioni compatibile per far lavorare tutte le unità funzionali, ma in quel caso potrà accedere solo che alcune

unità non lavorino; è necessario far sì che tutto questo accada il meno possibile. Ciò complica il compito del compilatore, ma semplifica di gran lunga l'hardware. Nell'HW di un VLIW non viene fatto alcun controllo sulle possibili dipendenze tra le istruzioni. La vera sfida è far sì che vengano minimizzati gli stalli. È il compilatore che deve trovare la giusta combinazione di istruzioni (all'interno della mega-istruzione) che minimizzino gli stalli: tipico esempio, il miss sulla cache. Se una delle 10 operazioni nella fase di scrittura dei risultati fa un miss nella cache dati stalla tutto fino a che quel miss non è gestito: si ferma tutto tranne quelle avanti alla stallata. Nei VLIW si hanno bisogno di molti registri per gestire:

- Loop unrolling
- Renaming
- Gestione degli hazard

Per alcune applicazioni i processori VLIW riescono ad avere prestazioni migliori rispetto ai superscalari, pur avendo una minore complessità dell'hardware ed un minore consumo di potenza!

Vengono usati in applicazioni special purpose, dove si sfrutta molto il calcolo parallelo, deve esserci un basso consumo di potenza a fronte anche di una minore quantità di silicio richiesta.

Si elenchino le varie tipologie di dipendenza che possono esistere tra le istruzioni, insieme con i possibili hazard che ne possono derivare

Una dipendenza tra due istruzioni è un legame tra due istruzioni ed è differente dal concetto di Hazard. L'ideale sarebbe avere tutte le istruzioni tra loro indipendenti, ma poiché è difficile che ciò si verifichi è necessario tener conto delle dipendenze al fine di ottimizzare il codice.

L'Hazard è la situazione in cui rischiamo di mandare in stallo la pipeline: ovviamente l'hazard dipende o deriva dalle dipendenze ma tengono conto di come è fatto il processore.

Le dipendenze possono essere cause di Hazard, e gli hazard dipendono non solo dal codice ma anche dal processore (e quindi dal fatto che il codice venga eseguito da un certo processore o meno).

- Dipendenze di dato: può essere diretta o indiretta (per proprietà transitiva A dipende da B, da cui a sua volta dipende da C). Si verifica quando un'istruzione produce un risultato che un'altra usa. Per il compilatore è estremamente semplice rilevare le dipendenze di dato che passano attraverso i registri, mentre è praticamente impossibile rilevare le dipendenze di dato che riguardano due celle di memoria: questo è un limite forte per le tecniche statiche. La tecnica dinamica invece lavorano run-time (a tempo di esecuzione) e quindi possono evitare questo fastidioso limite.

- Dipendenza di nome: si verifica quando due istruzioni fanno riferimento allo stesso registro p alla stessa locazione di memoria ma non c'è un flusso di dati associato al nome. Si distinguono in :

- Dipendenza di output
- Auto dipendenza

In pratica sono due istruzioni interferiscono tra loro se invertite senza un criterio che mantenga il risultato.

I possibili hazard che possono derivare da una dipendenza di dato sono 3:

- RAW: dovuta alla sovrapposizione delle istruzioni all'interno della pipeline, tipicamente per l'esistenza di una dipendenza di dato
- WAW: si possono verificare quando un'istruzione termina prima o in contemporanea ad un'istruzione che la precorre: dipendenze di output
- WAR: derivano dalle auto dipendenze

Si descriva brevemente che cosa sono e a che cosa servono le μ -ops introdotte da Intel a partire dal Pentium Pro

Sono delle micro-operazioni molto simili alle tipiche istruzioni RISC. Il processore traduce ogni istruzione IA-32 in una serie di μ -ops. Ad ogni colpo di clock vengono fetchate, decodificate e tradotte fino a 3 istruzioni IA-32. Le μ -ops vengono eseguite in una pipeline speculativa che usa register renaming e reorder buffer. Vengono eseguiti il renaming ed il dispatching alla reservation station di 3 μ -ops per colpo di clock. Può essere fatto il commit al massimo di 3 μ -ops per colpo di clock.

Si illustri brevemente da dove deriva il problema della consistenza delle cache, e come possa essere risolto

Il problema della consistenza della cache si ha quando in un sistema si ha più di una cache. Può succedere quando abbiamo cache L1 e L2 oppure quando è presente più di un processore con cache.

Per prevenire il problema della coerenza della cache viene adottato il protocollo MESI che è stato per primo implementato dal pentium.

Si basa sull'associare ad ogni linea di cache uno stato che può assumere 4 diversi stati:

- Modified (M): il dato nella linea di cache è disponibile in una sola cache dell'intero sistema. Il suo valore è diverso da quello che c'è in memoria. La linea può essere letta/scritta senza aver bisogno di accessi ad altre cache o alla memoria.

- Exclusive (E): il dato è presente in solo una cache dell'intero sistema. Il dato non è stato modificato fin ad ora, ed il valore è uguale a quello che c'è in memoria. Se occorre un'operazione di scrittura lo stato della linea di cache cambia in Modified.

- Shared (S): il dato nella linea di cache può essere contenuto in più cache del sistema. Ogni scrittura del dato (indipendentemente se si usa il write-through o il

write-back) deve essere fatta immediatamente in memoria, quindi le altre cache devono invalidare la corrispondente linea.

- Invalid (I): la linea non è logicamente disponibile in cache. La causa può essere che la linea è vuota o che contiene un dato non valido. Ogni accesso ad una linea invalida causa un miss. Una scrittura deve essere fatta in write-through, e la write-allocate non è supportata.

Il protocollo MESI fu sviluppato con il write-back e senza la write-allocate; si può estendere al caso di write-through ma in questo caso non esistono gli stati M e E.

Si illustrino brevemente le ragioni per cui alcuni processori ARM supportano il Thumb Instruction Set, e se ne descrivano le caratteristiche

Le istruzioni Thumb sono codificate su 16 bit anziché su 32 e sono identificate da un campo T nel CPRS (0: ARM instructions, 1: Thumb). Processando in thumb il processore diventa più lento, ma richiede un minore consumo di potenza (questo a fronte di un minore numero di accessi all'A-BUS). Inoltre, codificando in thumb, si ottengono molte più istruzioni ma con un codice che occupa meno spazio in memoria perché sono istruzioni più corte. I core che supportano il thumb hanno anche un decompressore che viene fornito incorporato e non si ha una perdita di prestazioni.

Si consideri l'architettura di Tomasulo: si illustri il funzionamento dell'unità load/store, con particolare riferimento al problema degli hazard RAW e WAR su celle di memoria

Se due operazioni di load e store si riferiscono allo stesso indirizzo X :

RAW -> bisogna evitare che una read all'indirizzo X venga eseguita prima di una write all'indirizzo X

WAR -> bisogna evitare di scrivere l'indirizzo X prima che la read abbia letto l'indirizzo X

Le unità load e store nel caso debbano aspettare il risultato di un'altra operazione occupano una reservation station come segue:

- un bit segna la reservation station come occupata
- Il campo op della reservation station tiene conto di quale operazione deve essere completata prima di poter procedere
- Il campo A contiene l'indirizzo da cui fare load o su cui fare store, prima è un indirizzo immediato, poi effettivo quando computabile

In questo modo controllando il campo A delle reservation station (una volta computato in ordine) è possibile evitare che 2 operazioni di load/store oppure store/load riferite allo stesso indirizzo di memoria vengano invertite di ordine, evitando quindi gli hazard RAW e WAR.

Si descriva brevemente la tecnica del data forwarding, utilizzata nei processori con pipeline per ridurre gli effetti degli hazard. Considerando la tipica architettura a 5 stadi (IF, ID, EX, MEM, WB) si elenchino le coppie di stadi tra cui è opportuno prevedere la logica di forward, descrivendo un caso di utilizzo per ciascuna situazione

Il problema del data Hazard nasce dal fatto che la ALU non ha ancora il valore corretto nel registro. Si fa quindi in modo che l'istruzione che ha bisogno di quel valore non lo prenda dal registro ma lo ricava dal registro a valle dell'ALU, ecco perché si chiama forwarding. Si ha bisogno della logica che riconosce l'Hazard. Il data forwarding non risolve tutti i problemi di data hazard a volte risulta necessario mandare comunque in stallo la pipeline. Le coppie di stadi in cui si rende necessaria la logica di forward è l'uscita dell'EX dell'istruzione precedente con l'ingresso della ID e della EX dell'istruzione successiva oppure dalla MEM della istruzione precedente e della ID e della EX dell'istruzione successiva.

Si descriva la sequenza di operazioni eseguita da un processore Intel che lavora in protected mode per il calcolo di ciascun indirizzo in memoria

Rispondo per quel che so dagli altri corsi : indirizzo generato dal programma (pagina|offset) -> la pagina indirizza la virtual page table estraendo l'indirizzo della pagina fisica -> controllo offset < limite pagina fisica -> se non presente in memoria si fa page fault per recuperarla -> accesso a pagina fisica + offset

Si descrivano la struttura ed il funzionamento del Branch Target Buffer

Il Branch Target Buffer è utilizzato in ambito delle tecniche di predizione dei salti.

Il Branch Target Buffer contiene per ogni riga:

- L'indirizzo di un salto eseguito in precedenza
- Il valore da caricare nel PC (e anche l'istruzione puntata nella versione ottimizzata)

Si accede al Branch Target Buffer tramite gli n bit bassi: confronto il campo lookup con l'indirizzo completo del salto, vado sul predicted PC per sapere il valore da caricare nel PC. Alla fine del campo c'è un bit che mi indica se il salto debba essere preso o non preso: non è particolarmente utile, perché caricherò nel PC comunque il campo predicted PC perché:

- Caso Taken ci sarà l'indirizzo a cui saltare
- Caso UnTaken ci sarà l'indirizzo successivo a quello di salto

Se dopo un accesso al BTB ottengo qualcosa, viene caricato nel PC l'indirizzo di salto. Dopodiché, nella decodifica si vede se il salto è stato preso o non preso. Se non viene preso, si svuota la pipeline e si aggiornano le righe oltre che caricare l'istruzione corretta.

Qualora non si ottenga nulla, si deve vedere se:

- non è un'istruzione di salto
- Il salto non è stato ancora eseguito, e l'entry viene aggiornata

- L'informazione riguardante quel salto è stata sovrascritta con l'esito di un altro salto avente gli stessi n bit bassi, il che può accadere quando in tabella esiste un salto diverso con gli stessi bit meno significativi.

Si descrivano le principali tecniche per realizzare il refresh delle memorie RAM dinamiche

Un ciclo di refresh è normalmente richiesto ogni da 1 a 16 ms, dipende dal tipo di RAM.

Sono usate normalmente 3 strategie per il refresh:

- RAS-only refresh: si basa sul mandare alla DRAM tutti i possibili indirizzi di riga senza inviare alcun indirizzo di colonna. In questo caso la DRAM legge e amplifica i valori di ogni riga, senza produrre alcun dato. Nei PC questo è portato dal canale 0 del DMA chip, che è periodicamente attivato dal contatore 1 del timer chip e fatto da un falso ciclo di lettura.
- CAS-before-RAS refresh: per le normali operazioni si attiva prima il segnale RAS (prendendo l'indirizzo di riga) e successivamente si attiva il segnale CAS. Per attivare questa logica, il segnale di CAS deve essere attivato per un certo tempo prima del RAS. si fa quindi il refresh di tutte le righe e si incrementa un contatore interno.
- Hidden Refresh: il ciclo di refresh è attivato dopo un ciclo di lettura tenendo il segnale CAS basso per un tempo lungo, e alternando su e giù solo il segnale di RAS. Questa tecnica permette di risparmiare tempo, il tempo richiesto per un ciclo di refresh è solitamente minore di un ciclo di lettura.

Si illustri brevemente il funzionamento dei predittori (m,n) nell'ambito delle tecniche di branch prediction

"m" rappresenta il numero di branches precedenti presi in considerazione per scegliere tra 2^m predizioni, ognuna delle quali è un "n"-bit predictor. Un m-bit shift register tiene in memoria la storia degli ultimi m salti (se presi o non presi).

Per effettuare il lookup della predizione si usa un indice unico formato dai bit meno significativi dell'indirizzo di salto uniti ai byte meno significativi del registro di m bit rappresentante l'esito degli ultimi m salti (la history).

Si descrivano brevemente i passaggi che portano all'indirizzo logico a quello lineare interno del meccanismo di segmentazione realizzato in protected mode dai processori 80x86

--

Si descrivano brevemente i 3 tipi di hazard che possono mandare in stallo una pipeline

- Structural Hazard: sono causati dal conflitto di risorse
- Data Hazard: un'istruzione necessita di un risultato elaborato dall'istruzione che la precede
- Control Hazard: dipendono dai salti condizionati nella pipeline

Gli hazard strutturali sono dovuti al fatto che uno stadio non riesce a completare in un tempo standard l'operazione e richiede più tempo. Oppure può accadere che al Register File venga richiesto un numero elevato di operazioni da svolgere. Altro caso in cui questo accade è il caso di cache miss.

La pipeline va in STALLO, e quando un'istruzione stalla:

- L'istruzione che parte dopo quella stallata andrà anch'essa in stallo
- L'istruzione che parte prima di quella stallata continuerà a lavorare.

Gli Hazard di Dato sono dovuti al fatto che le istruzioni sono parzialmente sovrapposte in fase di esecuzione.

I control Hazard, che sono dovuti ai salti. Condizionati: mentre valuto la condizione la pipeline continuerà a caricare istruzioni, ma se alla fine scopre che dovevo fare il salto dovrò svuotare la pipeline. Inoltre, è anche probabile che siano state effettuate operazioni di scrittura, il che fa degradare sensibilmente le prestazioni.

I control hazard creano un degradamento delle prestazioni molto più forte rispetto ai data hazard, perché causano lo svuotamento della pipeline.

Si definiscano gli hazard di tipo WAW, si illustri in quali architetture possono verificarsi

Gli hazard WAW (write after write) si possono verificare quando un'istruzione termina prima o in contemporanea ad un'istruzione che la precede: dipendenze di output.

Per evitarli bisogna controllare prima di far accedere allo stadio di EX un'istruzione dobbiamo controllare che questa non tenti di scrivere sullo stesso registro su cui sta già scrivendo un'istruzione si trova già sullo stadio di EX. In questo caso dovrei mandare in stallo nello stadio di MEM l'istruzione che è terminata in anticipo rispetto al flusso di codice: l'obiettivo di questa tecnica è far sì che le operazioni di struttura mantengano l'ordine che hanno nel codice.

Si possono verificare in architetture in cui ci sono più unità funzionali con latenze diverse. Questo causa che un'istruzione j , anche se successiva all'istruzione i , possa terminare la sua esecuzione prima. Se entrambe salvano il valore nello stesso registro ho una inconsistenza del valore del registro stesso.

Si descrivano le caratteristiche salienti del set di istruzioni dei processori ARM.

L'architettura ARM è un'architettura RISC che include:

- Architettura load/store
- Mancato supporto ad accessi alla memoria non allineati (supportati dal core v6)
- Set di istruzioni ortogonale
- Registri a 32 bit
- Opcode a lunghezza fissa per semplificare la decodifica e l'esecuzione a costo di diminuire la densità del codice
- Esecuzione in un ciclo di clock per la maggior parte delle istruzioni
- Esecuzione condizionata di molte istruzioni per ridurre i salti e compensare gli stalli della pipeline
- Le operazioni aritmetiche sono le uniche che possono alterare i registri delle esecuzioni condizionate
- Shifter a 32 bit che può essere utilizzato in contemporanea con la maggior parte delle istruzioni senza penalizzazioni di tempo
- Metodo di indirizzamento a indice molto potente.
- Interrupt a 2 livelli molto veloce e semplice con un sottosistema di registri collegati che commutano

Si descriva il meccanismo utilizzato dai processori ARM per la gestione delle eccezioni

Sono considerate eccezioni gli interrupt, trap e le chiamate da priorità superiori. Sono categorizzate in 3 gruppi:

- A effetto diretto sull'istruzione: interrupt sw, istruzioni indefinite, abort durante una prefetch
- A effetto collaterale: non sono generate dalle istruzioni ma comportano danni alle istruzioni (cache miss)
- Generate dall'esterno: (reset, irq, eiq)

Quando viene sollevata un'eccezione si completano le istruzioni precedentemente iniziate (se possibile), si passa al modo di funzionamento più appropriato, si salva il PC in r14, si salva CPSR in SPSR, disabilita gli interrupt sulle IRQ settando il bit 7 del CSR, disabilita i fast interrupt su FIQ settando il bit 6 del CSR (se l'eccezione è un fast interrupt) e forza il valore del PC tra 00 e 1C in base al tipo di eccezione. Questi valori sono gli indirizzi della vector address che contengono i salti agli exception handler.