

DIPENDENZE E ARCHITETTURE SUPERSCALARI

1

ILP: capacità di eseguire istruzioni multiple in // all'interno di un singolo flusso di istruzioni.

le tecniche per sfruttare l'ILP sono le pipeline, l'execution out of order, l'esecuzione speculativa

DLP: capacità di eseguire operazioni su + dati indipendenti in parallelo per migliorare le prestazioni del sistema. Molto sfruttato nelle arch. SIMD e vettoriali.

TLP: più thread simultaneamente → utilizzando più core nel processore. Sfrutta anche le risorse hw.

Due istruzioni sono INDEPENDENTI se si può invertire l'ordine delle istruzioni senza modificare il risultato.

DIPENDENZA DI NOME: se due istruzioni usano lo stesso nome ma non c'è flow di informazioni tra le due, non c'è una reale dipendenza.

- ANTI DIPENDENZA: se il registro usato come sorgente viene poi usato come destinazione non posso invertire l'ordine di esecuzione. Es. leggo X1 poi scrivo X1
- OUTPUT DEPENDANCE: un'istruzione dopo l'altra scrivono nello stesso registro.
~ Entrambe sono false dipendenze. ~

Per risolvere questi problemi si usa REGISTER RENAMING, ovvero possono essere riordinate o eseguite parallellamente le istruzioni se il nome viene cambiato per evitare il conflitto.

ALEE DI DATO

RAW: j prova a leggere una sorgente di dato prima che i lo scrive ed erroneamente legge il vecchio valore.

WAW: j scrive prima che i lo scrive, le scritte avviate nell'ordine sbagliato. Non ci sono problemi se il WB della 1^a avviene prima del WB della 2^a.

WAR: j scrive un operando prima che i lo legga, quindi i legge per sbaglio il valore nuovo.

SUPERPIPELINED PROCESSOR: In questo caso gli stadi della pipeline vengono a loro volta divisi in altre sottosezioni. Questo rende l'arch. + veloce.

(fck T) C'è un aumento delle estensioni e delle lunghezze delle pipe.

SUPERSCALAR PROCESSOR: attua ILP sfruttando + unità funz. in parallelo. Si eseguono + istruzioni in // se non hanno dipendenze. I due si possono mischiare.

Si può ottenere un buon compromesso aumentando il numero di u.f. nelle pipeline.

Se tutto fetchando e decodificando + di un istruzione per ciclo.

Possibilmente si eseguono due (o +) istruzioni (u // se si ha disponibilità HW e non ci sono dipendenze altrimenti si aspetta).

Pipeline Multiciclo

è in grado di fare commit ^(WB) out of order

Nelle pipe classica si sta 1Tck in ogni stadio, mentre nelle multiciclo ogni fase richiede uno o più cicli x essere completata.

$$\text{CPU Time} = (\text{N° istruzioni} \cdot \text{CPI senza stalli} + \text{N° stalli}) \cdot \text{Tck}$$

l'issue e l'execution vengono fatti in ordine, mentre il commit fuori ordine.

> le altre WAR non possono esserci, gli operandi sono sempre letti in ID quindi è impossibile che quando si legge un operando e l'istruzione successiva l'abbia già aggiornato.

> le altre WAW sono molto rare

Il completamento fuori ordine può dare origine a eccezioni imprecise es. se si genera un'eccezione a FDIW quando le istruzioni successive sono già state eseguite. Ciò bisogna di un HW + complicato. → Buffer che memorizza i risultati finché tutte le operazioni precedenti non hanno completato.

NOTA: una pipe gestisce le eccezioni in modo preciso se può essere fermata in modo che tutte le istruzioni precedenti all'istruzione che ha generato l'eccezione siano già state completate e le istruzioni successive non modificano lo stato delle CPU prima che l'istruzione sia servita.

NOTA: il WB out of order potrebbe portare alla richiesta simultanea di + WB in questo caso si stalla in ID.

Pipeline NON BLOCCANTI - OUT OF ORDER

Si vuole che un'operazione sia eseguita appena gli operandi sono disponibili anche se ad es. l'istruzione precedente è in stall. L'una istruzione successive può operarne una che è stallata in ID.

ID viene diviso in due:
- ISSUE: verifica se c'è disponibilità dell'unità fu necessaria
- READ OPERANDS: legge gli operandi dal RF.

Si fa issue in ordine ma si esegue fuori ordine non appena gli operandi sono disponibili. → Fetch e issue wonder, ex e wb out of order.

prima: ID → EX

ora: ISSUE → READ OPERANDS → EX

SCOREBOARDING: tecnica di scheduling dinamico che permette alla pipeline di modificare il flusso delle istruzioni. Si aggiungono blocchi HW che consentono di tenere parcheggiata un'istruzione in attesa che i suoi operandi siano disponibili. 3

Quindi lo scoreb. consente di eseguire le istruzioni out of order quando si hanno sufficienti risorse e no date dependencies. Execution out of order, WB out of order

→ C'è bisogno di un HW che tiene traccia dello stato dell'istruzione.

1. INSTRUCTION STATUS: informe su che stadio delle pipe si trova l'istruzione di cui è stato fatto l'issue. Permette di monitorare l'avanzamento dell'istruzione nelle pipe.

2. REGISTER RESULT STATUS: dice quale f.u. dovrà produrre un determinato ogni registro è associato a un dato. Tenne traccia dello stato dei registri del processore in campo che indica se il valore è già stato memorizzato; risultati delle operazioni.

3. FUNCTIONAL UNIT STATUS: ha una entry per u.f. e dice se questo è occupato cosa deve fare e quali sono gli operandi coinvolti.

Se fra una RAW, lo scoreb. legge tutte le istruzioni nello stadio delle pipe tenendo in pausa l'istruzione nella attesa che gli operandi siano disponibili.

STADI DELLA SCOREBOARD:

ISSUE, READ OPERANDS, EXECUTION, WRITE RESULTS.

RES: register result status: in ogni campo è riportato

tutte le istruzioni passate dallo stadio di issue in ordine, ma possono essere stallate in read operands ed entrare in execution out of order.

Perché nello Scoreboarding non c'è lo stadio MEMORY?

La gestione delle operazioni di accesso alla memoria (load / store) sono trattate in modo diverso, ovvero attraverso buffer di caricamento (load buffer) e salvataggio (store buffer) invece di passare attraverso uno stadio delle pipeline.

LOAD BUFFER E STORE BUFFER: le istruzioni L/S vengono eseguite in modo speculativo e memorizzate nei buffer di load/store. Le eventuali dipendenze vengono gestite nei buffer.

→ Non c'è bisogno di uno stadio dedicato nella pipeline.

TOMMASULO

4

fa scheduling dinamico e risolve problemi dello scoreb. facendo register renaming ed evita RAW e WAR.

evita antidependenze

ogni fu. dispone di una code di ingresso dove si parcheggiano le istruzioni in ciascuna u.f.

la RESERVATION STATION memorizza le istruzioni che devono essere eseguite dall'unità fu. associate in attesa che diventino liberi, gli operandi possono venire da registri del processore o da risultati di altre istruzioni in esecuzione o nella memoria. Inoltre la RS. tiene traccia dello stato di esecuzione delle istruzioni, ad es. può indicare se un'istruzione è pronta per l'esecuzione, o aspetta che gli operandi

diventino disponibili, se è in esecuzione o se è completa.
Nella RS ci sono 6 campi: • busy: se le es. è libera o no; Qj e Qk: dicono se il valore degli operandi è già disponibile o no
• op: operazione da eseguire; Vj e Vk: contenigono l'operando ($\exists Q_j \neq Q_k = 0$)

Nel register file affianco al valore contenuto nel registro c'è un bit di validità, se il valore che c'è non è valido viene riportato le RS che deve produrre quel valore.

(E.s. nell'esercizio d'esame era l'ultimo punto)

valid	tag v	tag v							
0	Mo.E	0	1						

DIFERENZA CON SCOREB: Tommasulo ha la RSS, albero vero e proprio "code" di attesa, scoreb. poteva contenere max 1 istruzione

Note: le RS sono occupate sia dall'istruzione in esecuzione sia da quelle ready. Nello studio di ID viene decodificata l'istruzione e viene allocata in uno determinate reg. se è disponibile, altrimenti si stallo in IO vengono risolte le RAW grazie al broadcasting dei risultati e tutte le unità in attesa degli operandi. Questo avviene grazie al common data bus, in cui quando le fu. finiscono l'esecuzione inviano il risultato e il tag associato all'operazione appena terminata.

X Mettere in esecuzione un'operazione è necessario che entrambi gli operandi siano validi e che l'u.f. sia libera.

Anche se RAW sono risolte mettendo nel RF il tag associato all'istruzione che produce il dato che serve.

LOAD-STORE → faccio issue delle load/store solo quando l'indirizzo è noto, se c'è una dipendenza di dato stallo in issue. Se è noto l'indirizzo si può usarlo come campo di tag nelle RS, che nell'caso load/store è il load store buffer.

se nel flusso di istruz. c'è un branch, finché non viene risolto, le istruzioni che vengono dopo di cui ho fatto issue non vanno in esecuzione ma rimangono nella res. st. [5]

Note: il controllo delle RS è libero mentre fatto nel 1° semip., mentre le WB avvengono nel 2° semip., per cui se anche si libera in quel ciclo le RS risultano occupate

Note: In Tomm. ci sono solo 4 stati: IF, ID, EX, WB (non MEM)

TOMMASO

VS

SCORSE.

⊕ Broadcast sul COB è + efficiente
(gli operandi sono disponibili
senza leggere dal R.F.)

⊕ Si evitano alle WAR

⊕ Si evitano alle WAW
rimanendo i registri
usando l'IO di una RS
anziché l'IO di un registro

⊕ RS x ogni u.f.

⊕ HW renaming of register
x evitare WAR e WAW

⊖ RS centralizzata

LOAD/STORE: gestite analogamente alle altre istruzioni, ove decodificate e inviate alle RS. Vengono però, durante l'esecuzione speculativa, usati buffer di caricamento e salvataggio x gestire le operazioni di accesso alla memoria. E' importante controllore dipendente e gestire l'ordine di accesso alle mem. x garantire coerenza.

In conclusione: TOMMASO stalle solo se alle strutturali

Quando ci sono dei branch si va avanti speculativamente nell'esecuzione ma i risultati non vengono salvati in memoria ma in un buffer finché non si avranno le certezze che veranno due eseguite. Nell'intervallo tra il completamento dell'esecuzione e la fase di commit il ROB e tutto il RF mette a disposizione i risultati.

I risultati nel register file li scrive il reorder buffer e non le reservation stations, perché tempo congelato nel ROB i risultati delle istruzioni finché non è stata fatta l'esecuzione in ordine delle istruzioni precedenti. [6]

Quando si fa issue si scrive un identificativo nell'entry del ROB che contiene l'istruzione che produrrà il risultato per quel registro.

→ Il commone dato bus vuol dire scrivere nel RF ma nel ROB

→ I dati arrivano dal ROB, dal RF, dal mem, dal cache

→ Bus di controllo: 32 bit → 32 bit

→ 32 bit → 32 bit Bus Machine

→ 32 bit → 32 bit Bus Memopipes

→ 32 bit → 32 bit Bus Regfile

→ 32 bit → 32 bit Bus ALU

→ 32 bit → 32 bit Bus Cache

→ 32 bit → 32 bit Bus Register

→ 32 bit → 32 bit Bus Memory

→ 32 bit → 32 bit Bus ALU

→ 32 bit → 32 bit Bus Cache

→ 32 bit → 32 bit Bus Register

→ 32 bit → 32 bit Bus Memory

→ 32 bit → 32 bit Bus Cache

→ 32 bit → 32 bit Bus Register

→ 32 bit → 32 bit Bus Memory

→ 32 bit → 32 bit Bus Cache

→ 32 bit → 32 bit Bus Register

→ 32 bit → 32 bit Bus Memory

→ 32 bit → 32 bit Bus Cache

→ 32 bit → 32 bit Bus Register

→ 32 bit → 32 bit Bus Memory

→ 32 bit → 32 bit Bus Cache

→ 32 bit → 32 bit Bus Register

→ 32 bit → 32 bit Bus Memory

→ 32 bit → 32 bit Bus Cache

→ 32 bit → 32 bit Bus Register

→ 32 bit → 32 bit Bus Memory

→ 32 bit → 32 bit Bus Cache

→ 32 bit → 32 bit Bus Register

→ 32 bit → 32 bit Bus Memory

→ 32 bit → 32 bit Bus Cache

→ 32 bit → 32 bit Bus Register

→ 32 bit → 32 bit Bus Memory

→ 32 bit → 32 bit Bus Cache

CACHE

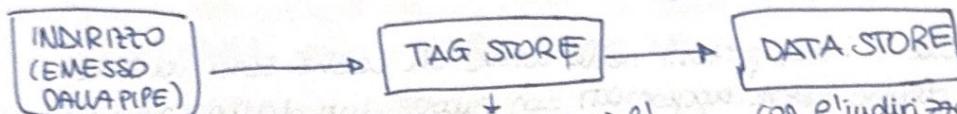
L2

Memoria basata su SRAM (+ veloce della DRAM). Si trova tra la load/store unit e riduce i tempi di accesso ai dati richiesti.

La cache sfrutta il concetto di località temporale, li salvano i dati usati + di recente. Fa risparmiare l'accesso alla DRAM che ha latenza.

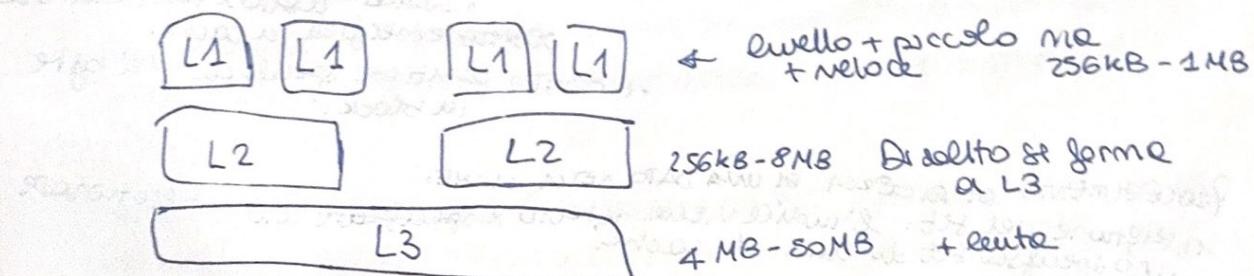
HIT: l'indirizzo è presente in cache

MISS: l'indirizzo non è presente in cache (il blocco viene poi portato in cache)



qui si controlla se c'è l'etichetta corrispondente all'indirizzo emesso.
Questo ha esito hit o miss

con l'indirizzo si entra nel data store per recuperare il dato cercato.



le cache moderne sono posizionate direttamente nelle CPU per ridurre le latenze.



Ese. se ho un blocco da 64 byte, questo corrisponde a 2^6 byte,
i 6 bit meno significativi identificano un offset nel blocco
i 6 bit più significativi, invece identificano un blocco in memoria

blocco di memoria: porzione delle memorie di dimensione fissata

np: cache che puo' immagazzinare 8 blocchi di dimensione 64 bytes
per identificare il blocco si guardano i bit + significativi che costituiscono l'etichetta.
+ free = 1 set ↑
1 blocco di mem. interamente in una linea

Set di cache: insieme di linee di cache. Quando un blocco di memoria deve essere memorizzato in cache, viene eseguito un algoritmo di mapping x decidere in quale set memorizzarlo. Una volta selezionato il set, il blocco di memoria viene memorizzato in una delle linee di cache all'interno di quel set.

la cache è composta da un insieme di blocchi di cache che contengono i dati memorizzati nella memoria principale. 13

Un blocco (o linea) di cache è costituito da:

• TAG: identifica univocamente i dati contenuti nelle linee di cache. È associato all'indirizzo di memoria del blocco di dati nella memoria principale.

• DATI: dati effettivamente memorizzati dalla memoria principale, costituiscono il contenuto delle linee di cache.

• VALID BIT: indice se i dati presenti nelle linee di cache sono validi o si devono essere aggiornati con nuovi dati dalla memoria.

Indirizzo di memoria diviso in: $\text{tag} = \text{bit} + \text{significativo}$
delle l'indirizzo di mem.
fisica utile x identificare
univocamente il blocco di dati
nelle cache.

• INDEX: usato x selezionare la posizione specifica in cache.

• OFFSET: posizione specifica del byte in block.

Procedimento di ricerca di un dato nella cache:

1) selezione del tag: l'indice viene usato x specificare uno specifico tag di tutte di cache.

2) confronto del tag.

CACHE DIRECTLY MAPPED: ~~ciascun blocco di memoria ha una posizione fissa e predefinita in cache.~~ (+ facile a livello hw ma può portare a conflitti) (ho bisogno solo di 1 comparatore x tutte le cache) (che compare direttamente il tag con l'indirizzo del mem. + se non trova il tag corrispondente al mem. si legge da un altro set)

Fully associative: i blocchi di memoria possono andare in una qualsiasi posizione nelle cache. (HW più complesso, no collisioni) (No bit di indice)

SET ASSOCIATIVE: ogni set ha + linee di cache e un blocco di mem. può essere memorizzato in qualsiasi linea nel set. (tag + offset) x cui ho due comparatori, il tag può trovarsi in due possibili blocchi di cache (o ho 2 vie) sull'indice identifica il bit di indice: $\log_2 n$ blochi di cache o 0 se fully ass.

Quando si ricerca un indirizzo in cache si cerca prima tramite bit di indice poi di tag e poi offset, poi si controlla il valid bit

Se il blocco si trova in cache (hit), il tag memorizzato deve essere valido e corrispondere al tag del blocco x il quale ho fatto l'accesso.

n° bit x identificare le byte in block: \log_2 dimensione del blocco

- ⊗ Due blocchi in memoria mappati allo stesso indice della cache non possono essere presenti contemporaneamente nella cache.
- Può portare a una hit rate dello 0% se si accede alternativamente a uno stesso blocco mappato allo stesso indice. (MISS o CONFLITTO)

l'associatività / hit rate (un blocco non ha una sola posizione in cui può stare, ma de 2 (set associativo) fino a in tutta la cache con fully ass.)
l'hw costoso (ho bisogno di + comparatori) / latenza di accesso alle cache.

set composto da un solo blocco: cache directly mapped

set composto da + blocchi (ma meno delle totalità): set associative

set composto da tutti i blocchi che compongono il dato store: fully associative

RECAP: struttura delle CACHE: può essere organizzata in più set, ognuno dei quali contiene un certo numero di linee di cache.

Access

Associatività: determinare il n° di linee di cache in ciascun set.

Ese. associatività 4: ogni set contiene 4 linee di cache.

COSTRUZIONE DELLE SCRITTURE NELLA CACHE

12
5

- **WRITE THROUGH**: nel momento in cui avviene la scrittura viene aggiornata subito la main memory e i livelli successivi alle cache.
- **WRITE BACK**: scrivo solo in cache e aggiorno la memoria solo quando rimuovo il blocco dalla cache. → Risparmio energetico richiede un bit che dice, nel tag store, se il blocco è sporco.
Se il blocco è sporco.
- **WRITE ALLOCATE**: tutte le miss in scrittura necessitano di allocare il blocco letto nella main in cache
Le write miss sono trattate come le read miss.
- **WRITE AROUND**: scrive direttamente in memoria senza portare in cache

LEVEL DI CACHE

- first level caches: piccole, bassa associatività, lettura e critica tag store e dato store acceduti in parallelismo.
- second level caches: high associativity tag store e dato store acceduti in serie.

TERMINOLOGIA

1 CAPACITÀ: Numero di byte che la cache può immagazzinare (dato store)

2 DIMENSIONE BLOCCO/LINEA: bytes portati in cache in una sola volta

3 NUMERO DI BLOCCI: numero di blocchi nelle cache ($1 \div 2$)

4 GRADO DI ASSOCIAZIONE: (numero di vie) numero di blocchi in un set

5 NUMERO DI SETS (3/4)

n^o bit x identificare un set: $\log_2(n^o \text{ set})$ es. 128 set $\rightarrow \log_2(128) = 7 \text{ bit}$

$$n^o \text{ set} = \frac{n^o \text{ blocchi}}{n^o \text{ vie}}$$

CAPACITY MISS

La cache è troppo piccola x contenere tutti i dati di interesse contemporaneamente. Se la cache c'è piena e il programma vuole leggere il dato x che non è presente in cache, cancella (evict) il dato y e ci mette x.

Se il programma vuole di nuovo accedere al dato y c'è capacity miss e se dato x sarà collocato in set particolare. Come scegliere y x ridurre al minimo la possibilità di avere di nuovo bisogno? LRU

If set c'è un LRU bit

(in un set ci sono tante linee di cache quante sono le vie)
Spreco banda pochi blocchi non sfruttano bene l'energia
e lo spazio tempo non sfrutta bene la località spaziale

DIMENSIONE DEL BLOCCO? CONVIENE GRANDI o PICCOLI?

NOTA: il blocco è una quantità di byte di mem. portati in cache, all'interno dei quali si naviga col byte in block.

CLASSIFICAZIONE DELLE MSS

- MISS OBBLIGATORIE se è la 1^a volta che si accede al dato.
- MISS DI CAPACITÀ cache troppo piccola (quelle che non sono le prime) di accessi successivi al primo falliscono perché il n° di vie è troppo piccolo.
- MISS DI CONFLITTO (quelle che non sono le prime) di accessi successivi al primo falliscono perché il n° di vie è troppo piccolo.

Solutions:

→ Sol: prefetching

→ > associatività,

CACHE MULTICORE

Cache → privata

→ condivisa: > capacità, + facile mantenere coerenza, + bassa latenza

CONSISTENZA

COERENZA:

cache coherence problem: nasce quando una cache vuole modificare un dato che è presente già in altre cache. Come ti garantisce che tutti i processori leggendo vedono lo stesso valore aggiornato?

Il dato scritto potrebbe essere inviato alle altre cache e farle aggiornare (invio dato e registro) o la si può invalidare (invio solo il registro).

SNOOPY BUS: bus condiviso dai cache controller, che garantisce sincronizzazione: i cache controller sono in ascolto relativamente alle modifiche che fanno le altre cache.

c'è un blocco di cache che in metadato di coerenza
le cache prima di effettuare una lettura o scrittura consulta il directory e vede se qualcuno ha modificato il dato. In quel caso il directory comunicherà a quelli di validare e salvare il valore nel memoria così che chi ha fatto richiesta di accesso abbia il dato corretto.

con WRITE AROUND: se ho miss in scrittura modifco direttamente la main memory senza portare il blocco in cache. Se usassi write allocate in caso di miss in scrittura avrei una copia locale corretta ma la main m. non aggiornata.

PROTOCOLLO VALID / INVALID: un altro blocco di sincronismo tra cache. Se uno ne vuole modificare un dato deve comunicare sul bus e' udito e le altre cache se hanno una copia lo invalidano. In scrittura → write through miss in scrittura → write around

PROTOCOLLO MSI

L7

Si Nasce dal desiderio di usare le protocollo write back x cui ce' bisogno di aggiungere uno stato: MODIFIEP, ovvero bisogna informare che il dato che e' appena stato scritto ha portato i^e blocos in uno stato # de quelli che c'e' in memoria e che quest'ultime non ha il dato aggiornato. x cui se viene richiesta da un altro P una lettura del quel blocco bisogna fare un ciclo di WB (x aggiornare le main) e poi si va in shared.

Le linee di cache si puo' trovare in M, S, I.
 + non si ha quel dato update
 + copie locali
 copie locali non validi entro
 accerente con le main e l'unica

NOTA: quando si fa una read e si ha null e si prende dalle main M. si va direttamente in shared, ma ci e' lo stato E.

PROTOKOLO MESI

Quando si fa richiesta di load sul bus bisogna vedere se quel dato e' già in altre cache, in questo caso si va in shared, se invece non c'e' già in altre cache si va in Exclusive. ($I \rightarrow E$)

PRO: si passa da E a M senza far sapere nulle e nessuno tanto sono sicuro che in cache ho l'unica copia.

Se mitto in shared e il processore mette uno state, prima di tenere veramente una lettura esclusiva in modo da invalidare le altre copie x poi modificare il mio dato e andare in M.

Se sono in M e un altro cache vuole fare read & read exclusive faccio blash, ovvero aggiorno la main memory con un ciclo di WB. A questo punto da M vedo in S se le richieste era di lettura

e vado in I se le richieste era read exclusive.

PROBLEMA DI MESI: ogni volta che una linea e' in M, se un altro vuole leggere deve aggiornare il valore nella main memory, e questo potrebbe volutare quasi infinito xxe' e breve qualcun altro lo risolvono?

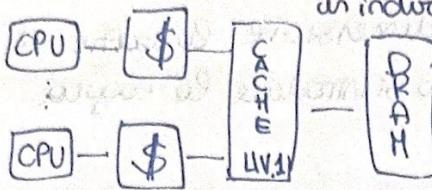
→ Introdurre O: OWNER

→ Introduzione di un bit per dire chi e' l'unico proprietario di quel dato

→

MEMORIA VIRTUALE

→ consente ai programmi di usare + memoria
di questo effettivamente disponibile. 1



E' un'astrazione che consente ai programmi di vedere un indirizzamento virtuale continuo indipendente dalla mem. fisica
contiene: stack, codici dei programmi
sistema operativo, dati del S.O.

Es. 8GB DRAM → dovrei preallocare il modo statico tutte le cose
Questa è una grande limitazione (ma va bene x sistemi embedded)

Voglio garantire che un programma possa accedere solo a un set di indirizzi. Per garantire questo ci sono 2 approcci

Solo MEMORIA FISICA

(es. embedded, senza S.O.)

Qui le load/store accedono
direttamente alla mem. fisica

Bisogna stare attenti all'

ISOLAMENTO

Il software da solo non può garantire
che gli accessi vengano fatti solo dove
è consentito. Meglio, c'etterebbe molti
tacchi alla memoria.

COME GARANTIRE ISOLAMENTO:

• SEGMENTAZIONE: presenza di registri mappati in memoria che specificare
regioni di indirizzi a cui un determinato programma può o meno accedere.
Sarà l'HW a controllare gli accessi.

PROBLEMA: frammentazione

9

PAGING:

suddivisione in sette (pagine) di uguale dimensione lo spazio di indirizzamento virtuale e fisico. Nel mezzo si introduce la logica di traduzione.

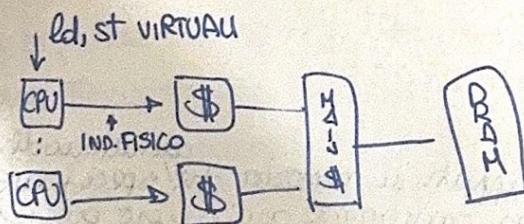
OVERPROVISIONING: Si può avere all'interno di un sistema molte più applicazioni che usano più dati rispetto alla memoria fisica disponibile.

Tutte le pagine di memoria virtuale non staranno interamente all'interno delle memorie fisiche.

Si risolve così: le pagine di memoria virtuale associate ai processi che non possono stare in un determinato momento in memoria fisica vengono salvate nel disco. Quando un programma richiede quelle pagine di mem. virtuale le si preleva dal disco e le si mette in memoria fisica, se non c'è spazio si sposta nel disco qualcosa altro. (SWAPPING)
si rimuove una pagina di memoria virtuale che era precedentemente allocata in memoria fisica, nel disco, e dare spazio a altre pagine di mem. virtuale)

⇒ LA MEMORIA VIRTUALE È UN'ABSTRAZIONE CHE CI CONSENTE DI ALLOCARE POTENTIALMENTE INFINTA MEMORIA NEI NOSTRI PROGRAMMI, PUR AVENDO UNA MEMORIA FISICA

costo: processo di traduzione (tempo, logico)



In ingresso alle CPU: indirizzo virtuale

In uscita dalla CPU: indirizzo fisico

PAGE TABLE: grande tabella con 1 entry per pagina di memoria virtuale e come uscite ci sarà una pagina di memoria fisica.
Si entra con l'indirizzo virtuale che il processore ha emesso e l'indice della pagina associata all'entry della tabella si trova il corrispondente indice di pag. in mem-fisica.

- le pag. di mem. fisica < pag. mem. virtuale ⇒ non tutte le pagine di mem. virt. hanno contemporaneamente una traduzione valida

PAGE FAULT: la pagina di mem. virtuale associata all'indirizzo di cui ho richiesto l'accesso, non ha una traduzione valida verso la memoria. Un interrupt service routine del sistema operativo interverrà e troverà una pagina di mem. fisica libera, altrimenti fa SWAPPING

• PAGE SIZE: dimensione della regione di memoria che condivide la stessa traduzione degli indirizzi.
E' una grandezza fissa, generalmente 4 kibyte 3

• ADDRESS TRANSLATION: procedura con cui si ottiene l'indirizzo fisico a partire da quello virtuale

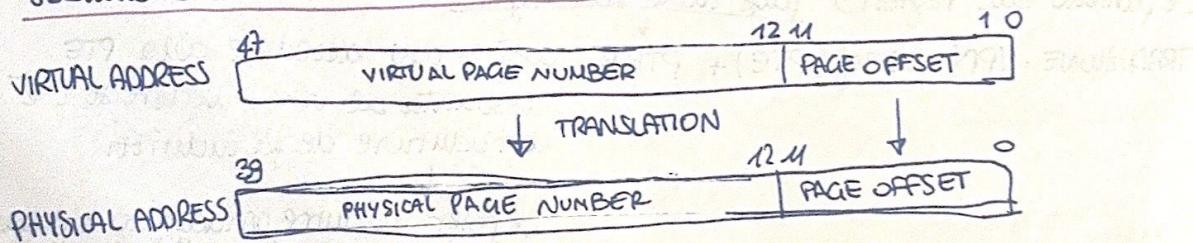
• PAGE TABLE: Tabella che serve per trovare il corrispettivo indirizzo fisico di un indirizzo virtuale. Non contiene la traduzione di un singolo indirizzo di mem. virtuale rispetto un singolo indirizzo di mem. fisica, ma le pagine di memoria fisica associate alle pag. di mem. virtuale.

N° PAGINA MEM. VIRTUALE → N° PAGINA MEM. FISICA

MMU: MEMORY MANAGEMENT UNIT: si occupa delle traduzioni, è un bus e si appoggia alle page table, che è gestita dal s.o. e sfrutta il TLB (translation lookaside buffer) che accelera la traduzione.

Disposito (Intel): 48 bit di spazio indirizzamento e ottimo ai 40 bit di spazio di indirizzamento fisico.

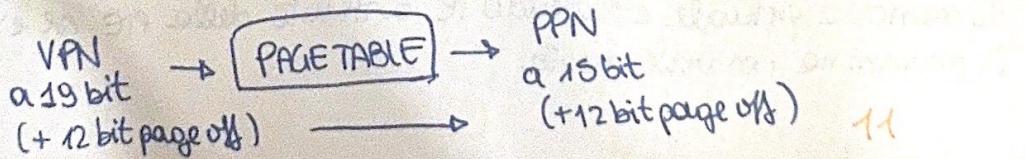
traduzione da indirizzo virtuale a fisico



- I 12 bit meno significativi identificano il PAGE OFFSET (sempre) e sono uguali sia nell'indirizzo di mem. virtuale che fisica.
- I restanti bit + significativi vengono tradotti nel physical p. n.

ESEMPIO:

- Dimensione memoria virtuale: $2GB = 2^{31}$ bytes → Indirizzo virtuale: 31 bit
- Dimensione memoria fisica: $128MB = 2^{27}$ bytes → Indirizzo fisico: 27 bit
- Dimensione della pagina: $4kib = 2^{12}$ bytes → Page offset: 12 bit
- N° pagine mem. virtuale? $2^{19} (2^{31}/2^{12}) \rightarrow$ Indice di pag. mem. virt.: 19 bit
- N° pagine mem. fisica? $2^{15} (2^{27}/2^{12}) \rightarrow$ Indice phys.: 15 bit



niga della
Una page table contiene una entry per pagina di mem. virtuale.

4

contiene:

- valid bit (se esiste un physical page number associato all'indirizzo virtuale vale 1, se 0 page fault)
- physical page number
- bit aggiuntivi o.s. dirty bit (se la pg. virt è stata modificata rispetto a quando è stata caricata)
- bit di replacement (es. LRU) bit di protezione.

Nota: una pag di mem fisica può essere usata da più spazi di memoria virtuale associati a processi diversi.

Esempio: libreria → ci saranno più indirizzi di mem. virtuale associati a + processi che rimappano sulla stessa pagina di memoria fisica.

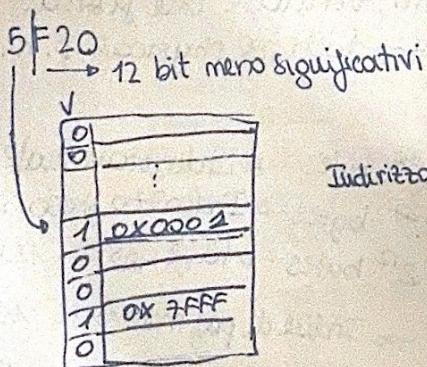
✓ Da quanti bit è composta una page table entry? deve contenere PPN e vari bit

✓ Quante page table entry ci sono in una page table? 2^{19} (512 kpte) ovvero tante (In rig. all'esempio) pte quante sono le pagine di mem. virtuale

. Dove si trova la page table? In DRAM a partire da un certo indirizzo base che è puntato dal registro page table base register.

TRADUZIONE: VPN · sizeof(PTE) + PTBR così si può accedere alla PTE associata al VPN e vedere se c'è traduzione degli indirizzi

Esempio: ld x2, 0x5F20 (x0)



Indirizzo fisico: 0x0001F20

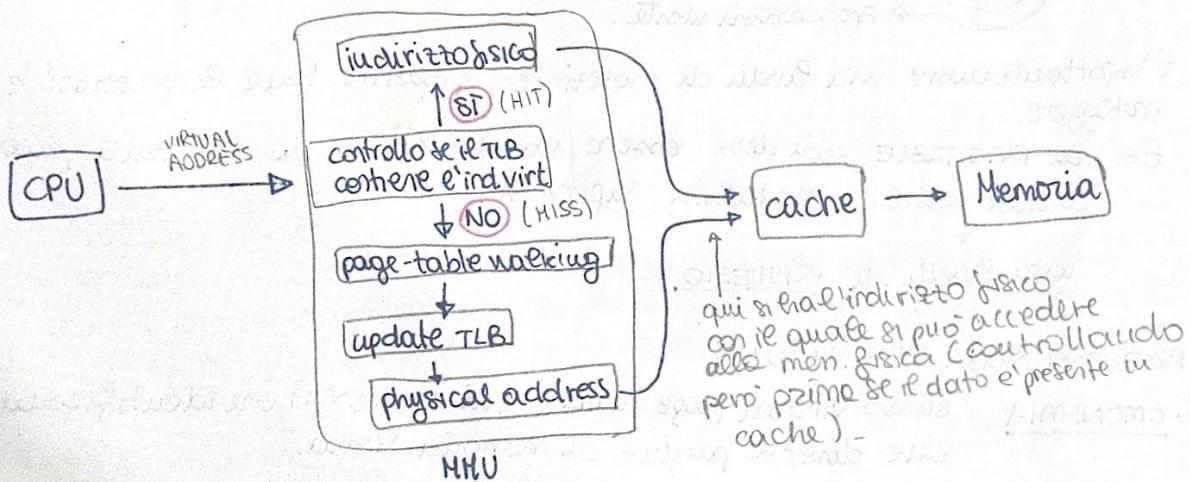
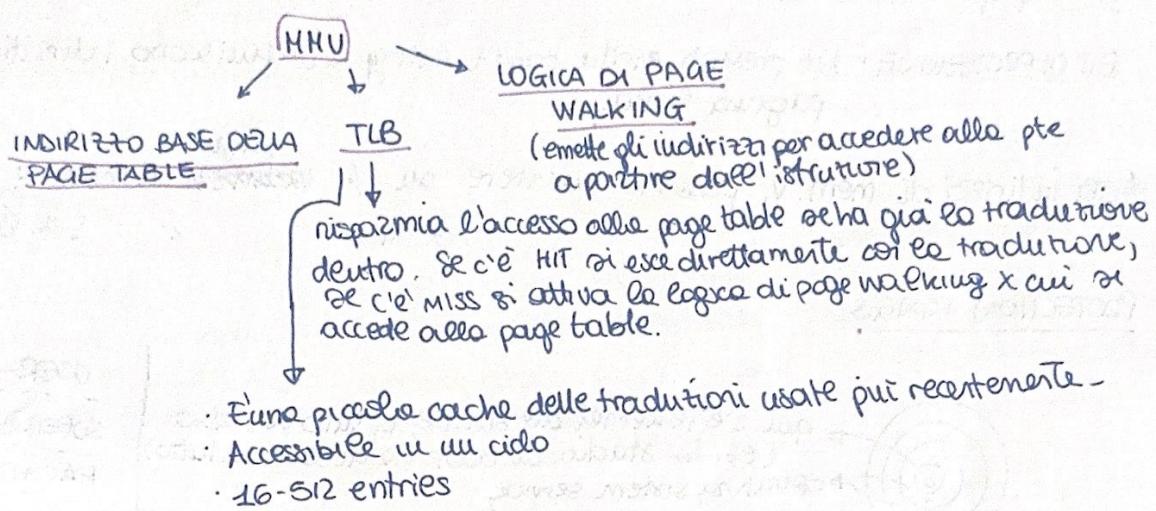
per tradurre un indirizzo bisogna accedere in memoria x andare alle page table, prenderne il contenuto e comportarlo con il page offset e a quel punto accedere all'indirizzo corretto.

PAGE WALKING: logica contenuta nello MMU, scomponere l'indirizzo di memoria virtuale e mettendo il contenuto delle PTE se è valida. Il programma non vede questo.

MULTI-LEVEL PAGE TABLE

Di solito si hanno più livelli. Si divide la page table in sottosezioni e se ne tiene in memoria solo una sottosezione. I bit + significativi dell'indirizzo di memoria virtuale definiscono le entry a una page table di livello 1 con i rimanenti bit + significativi si identifica un indirizzo che punta a una page table di 2° livello che contiene le traduzioni.

Page table a + livelli → + accessi in memoria



MEMORY PROTECTION

6

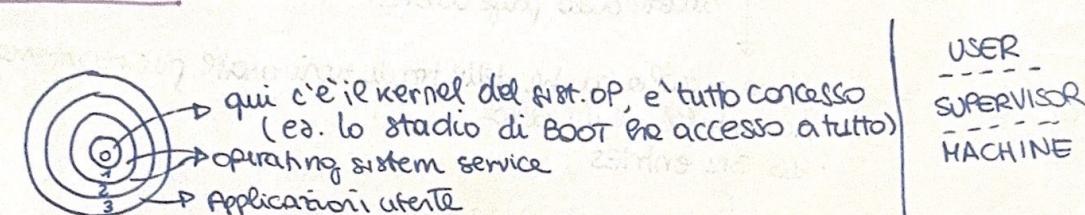
E' importante proteggere zone di memoria fisica da letture/modifiche da parte di programmi che non hanno il diritto di farlo.
(spesso più programmi eseguono contemporaneamente)

- Ogni programma ha a disposizione l'intero spazio di indirizzamento virtuale e una propria page table. ~~MA~~ può accedere solo alle pagine fisiche mappate nella propria tabella.

BIT DI PROTEZIONE: bit presenti nella page table entry e definiscono i diritti sulla pagina fisica.

~~gli~~ indirizzi di mem. V. possono riferire su ~~le~~ pagine di mem. F.
o su = // (es. librerie)

PROTECTION RINGS



E' importante avere vari livelli di protezione, x gestire bene le eccezioni e proteggere.

Esempio: la PAGE TABLE non deve essere modificabile da un utente, ma lo deve essere in modalità supervisor

+
vari livelli di PRIVILEGIO.

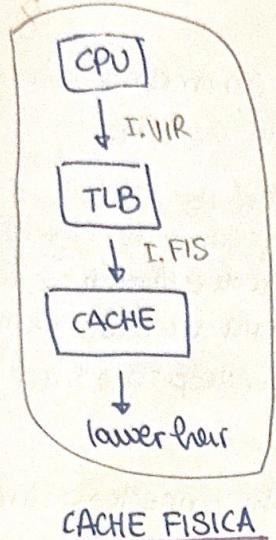
PROBLEMI NELLA MEM. VIRTUALE:

- OMONIMIA: stesso virtual page number (di +processi) che identificano due diverse pagine di memoria fisica.
- SINONIMIA: diversi virtual page number che identificano la stessa pagina di memoria fisica.

Da qui: dove posizionare la cache per avere un ottimo compromesso fra ottimizzazione ed evitare problemi creati da omonimi e sinonimi.
ovvero: cache indirizzata

↓ ↓
FISICAMENTE VIRTUALMENTE

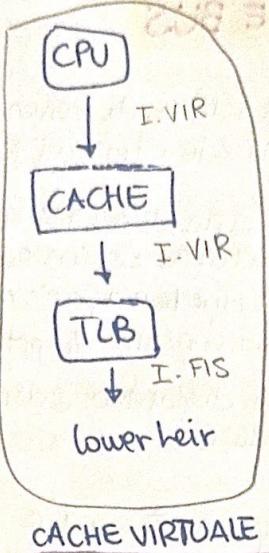
Quindi si mette prima le TUB o le cache?



CACHE FISICA

Si accede in cache dopo aver fatto la traduzione dalla MMU (o TLB).

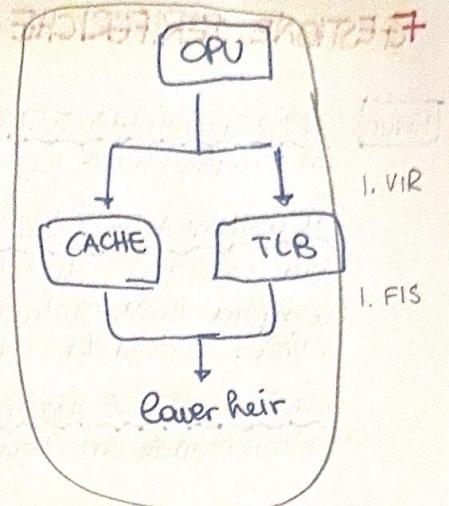
Non ci hanno problemi di omoniimi e sinonimi perché in cache si eutra con l'indirizzo fisico.



CACHE VIRTUALE

Se si fa hit si risparmia la traduzione.

c'è il problema dei sinonimi e degli omoniimi, quando si cambia processo si dovrebbero invalidare le cache. il tag avendo bit di indirizzo virtuale, cambiando programma, ma potrebbe esserci omomimia.



MIX : VIPT

Si accede ai bit di set ID con bit virtuali e ai bit di tag con l'indirizzo fisico.
(virtual index e physical tag)

- ISTRUZIONI CSR: Istruzioni dedicate che possono leggere/ modificare specifici bit di un registro che hanno un significato particolare. ed. bit di privilegio, gestione interrupt.

→ All'interno della MMU:

la MMU è condivisa dallo stadio di fetch e dallo stadio di mem, si fanno due TLB diversi, uno per i dati e uno per gli indirizzi delle istruzioni.

In un ciclo puo' essere servita una miss nel TLB dati e una nel TLB address.

La MMU è condivisa dallo stadio fetch e mem.

Se si fa hit nel tlb si può accedere contemporaneamente ai dati e alle istruzioni, e non avere stalli strutturali. La logica di page walkup è invece condivisa da dati e istruzioni, se si ha una miss nel tlb per entrambi, la ricerca della pagina di mem. fisica sarà condivisa e verrà eseguita.

GESTIONE PERIFERICHE - DMA E BUS

8

Heur. SPAZIO DI INDIRIZZAMENTO = range di indirizzi di memoria usati da un processo o da un S.O. per accedere e gestire i dati e le istruzioni memorizzate in memoria.

BUS CON LOGICA THREE STATE = consente a più dispositivi di condividere lo stesso bus dati, indirizzo e di controllo. La logica t.s. consente a un dispositivo di spegnere la propria uscita, rendendola alta impedendo, ciò consente ad altri di collegarsi al bus. Contro: costosa da implementare, + consumo di potenza, limitata velocità di trasmissione.

INTERCONNESSIONE PUNTO PUNTO = ogni dispositivo è direttamente collegato ad un altro tramite un collegamento dedicato.

Oggi le connessioni vengono fatte tipicamente con bus sincroni che connettono due punti di comunicazione in cui uno è master e l'altro slave.

BUS SINCRONO = la comunicazione su un bus sincrono avviene in modo coordinato da un segnale di clock che sincronizza l'arrivo e la partenza tra due dispositivi collegati al bus.

FUNZIONAMENTO DEL BUS

Segnale di handshake: viene tenuto alto da chi vuole iniziare una comunicazione.

Nel caso sia una operazione di READ, chi inizia la comunicazione deve generare l'handshake ed è READ ENABLE e si mette sul filo l'indirizzo di cui si vuole fare la lettura. Ci sono due fili unidirezionali (x uscite logica t.s.) uno dal processore alla memoria e uno opposto. Ci sarà anche un canale di handshake acknowledge gestito dallo slave verso il master.

Qualndo il handshake e l'acknowledge sono entrambi alti, al successivo fronte di clock inizia la comunicazione.

CASO: 1 INITIATORE, MULTIPLEX TARGET

Il canale degli indirizzi è pilotato dal dispositivo che inizia la comunicazione ed è condiviso a tutte le porte target dei dispositivi connessi al bus.

LOGICA DECODICE = connette elettricamente i segnali di write enable e read enable emessi dal master, in modo selettivo ai singoli dispositivi.

Nel più possibile, senza logica t.s. connettere 3 uscite allo stesso ingresso, mentre si può connettere un'uscita a più ingressi.

CASO: PIÙ INITIATORI

c'è bisogno di arbitraggio che serializza le richieste. Con cache di demux solo un master verrà abilitato. Inoltre dato che si può avere un'uscita connessa a + ingressi, il segnale di risposta è connesso sia al device (master) che al device che ha fatto la richiesta che aveva iniziato la comunicazione vedrà nel segnale di ack.

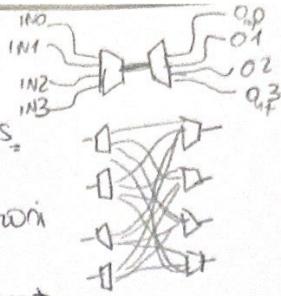
↓
COMUNICAZIONE BLOCCANTE

CROSSBAR: si hanno + initiatori e + target

Si vuole consentire + comunicazioni contemporaneamente

Il blocco crossbar è il più usato nei SoC, lo stesso dispositivo può avere più porte in ingresso e in uscita.

INTERRUPT CONTROLLER: componente HW che gestisce le interruzioni provenienti da dispositivi HW esterni o interni.
Consente di definire le priorità sulle linee degli interrupt.



9

DHAC: componente HW che gestisce trasferimenti di dati diretti tra dispositivi periferici e la memoria principale senza l'intervento della CPU.
Le trasmissioni avvengono ad alta velocità, non incidono sulla CPU che si concentra su altre attività computazionali, diminuisce la latenza

cosa succede se si introduce la mem. virtuale alle periferiche

+ parleremo di iud. fiscia

i cores parlano a iud. virtuali che ne fanno tradotti.

Quando si vuole comunicare con una periferica c'è di mezzo un device driver che consente la comunicazione.

Al device driver viene però comunicato un indirizzo virtuale che lo comunica alla periferica che però parla i indirizzi fisici.
Quando la pr. accede in memoria non accede all'indirizzo fisico corrispondente a quello virtuale, ma scrive direttamente messo dal dev. dr.

attraverso l'ind. virtuale pensando sia fisico e sbaglia.

Anche i controlli di protezione non sono applicati alle periferiche.

Ma esteso l'HW.

Anche gli I/O hanno ora una loro MMU che consente di tradurre gli indirizzi e avere protezione.
Questo crea la unified memory, cioè gli I/O possono condividere e usare la stessa DRAM, ce ne voleva una apposta x evitare i problemi sopra.

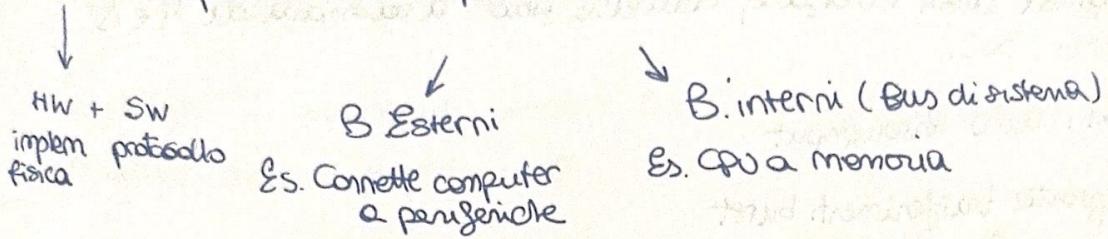
La IOMMU contiene TLB e page walking, quando c'è page fault viene comunque gestito dal processore.

Nei casi + sofisticati anche la IOMMU ha le PT walking.

AMBA AXI

10

Bus: sistema di comunicazione che consente di trasferire dati tra diversi componenti in un computer



3 TIPI DI SEGNALI: DATA BUS

ADDRESS BUS: x selezionare la periferica

CONTROL SIGNALS: sincronizzare e identificare trasferimenti

2 TIPI DI BUS

BLOCCANTI

NON BLOCCANTI

sono operazioni atomiche → consentono di avere + richieste simultanee

le linee su cui passano i dati tra M e S non sono condivise ma per identificare il target della comunicazione; quando il master emette l'indirizzo, questo entra in un blocco di decodifica (address decoder)

TIPICA OPERAZIONE BUS

1. Il master seleziona una periferica comunicando l'address all'address bus esteso e segnale di controllo ad es. su Read/Write.

NOTA: il parallelismo di un bus (es. 32 bit) non impedisce la sua lettura a quel valore, si possono fare 64 bit leggendo 32 alle volte. Il // influenza le velocità del trasferimento.

2. Quando lo slave è pronto manda il dato richiesto e segnale ~~ready~~ ^{valid} inoltre invia il segnale di ready sul bus

3. Il master legge il dato

4. Può inviare una nuova richiesta

Perché c'è bisogno di una comunicazione standard?

Design modulare

Consente il riutilizzo del design

AMBA 5 è un protocollo di interconnessione on-chip
è molto usato in PC, tablet, telefoni.

10
11

AXI: advanced extensible interface

E' la + diffusa AMBA interface, connette fino a centinaia di MeS
in complessi SoC.

AMBA 3 AXI: alto throughput

AXI 4: supporta trasferimenti burst

AXI 4 Lite: non lo supporta

AXI STREAM: riduzione del protocollo AXI in cui non esiste il canale degli indirizzi

TRANSAZIONE: insieme di operazioni necessarie a effettuare una comunicazione

Il protocollo AXI è burst e ha 5 canali indipendenti

c'è separazione tra canali di lettura e scrittura x indirizzi e dati
read address, write address, read data, write data, write response

S comunica dati eth a M M vuole scrivere dati S bus canale consente dati S x dare a M che la scrittura è terminata

ARCHITETTURE VETTORIALI & SIMD

Esistono due macro famiglie di architetture che consentono di fare la stessa istruzione su più dati, ovvero gli array processor (SIMD) e vector processor.

Il primo è un'architettura di tipo array che opera sui multipli dati nello stesso tempo su multiple unità funzionali, mentre i vector processor sono acceleratori che eseguono operazioni su multipli dati nel tempo condividendo lo stesso HW.

I usati oggi sono un ibrido dei due.

VLEN/VECTOR LENGTH: indica la lunghezza dell'array, rappresenta il numero di elementi che possono essere elaborati contemporaneamente in un singolo ciclo di istruzione vettoriale.
es. se VLEN = 256, l'architett.vett. può eseguire operazioni su 256 elementi contemporaneamente.

VLR: registro specifico usato per contenere le lunghezze dei vettori che verranno elaborati durante un'operazione vettoriale.

$$VLR \leq VLEN$$

\Rightarrow VLEN definisce le lunghezze max supportate dai vettori delle architecture vett., mentre VLR è un registro che può essere usato per specificare la lunghezza effettiva usata in un det. contesto.

VECTOR REGISTER FILE: ha un tot di registri, ad es. 32
di lunghezza fissa che può essere 64, 256, 512...
(fusano: 512 bit)

VETTI

La lunghezza di un registro vettoriale può essere 64, 256, 512 bit

Suppongo ~~settore~~ di lunghezza 512 bit
registro

Suppongo di avere un vettore con 8 elementi da 64 bit
ognuno, a pieno perfettamente.

VLR: indice al processore quanti elementi di un vettore devono essere elaborati durante un'operazione vettoriale contemporaneamente.

Prima di eseguire un'operazione vettoriale, il software imposta il valore desiderato nel VLR

Ese. se $VLR = 8$ il processore eseguirà l'operazione su 8 elementi del vettore contemporaneamente.

(gestisce il parallelismo)

$VLR \leq$ disponibilità microarchitetturale

NOTA: con l'arch. vett. si risparmiano i getch. Es. quando si fa Vec. Load. Supponiamo che si deve accedere ai successivi x elementi a partire da quell'indirizzo utile → prefetching

VECTOR REGISTER FILE & SCALAR REGISTER FILE

32 registri vettoriali
con 32 registri interni

- con sicurezza:
- VLR
 - registro di configurazione
che dice quanto elementi
è lungo un vettore ?

Ese. WAD VETTO RIALE

l'indirizzo base è scalare ed è contenuto in un registro scalare.
il registro dest. è vettoriale poi
registro scalare che dice di quanto
deve essere incrementato l'indirizzo
di un elemento al successivo
nel vettore.

Il VLR restituisce la posizione del
registro vettoriale $\rightarrow VLR \leq$ registro vettoriale

for ($i=0, i < 64, i++$)
 $c[i] = A[i] + B[i];$

SCALARE	VETTORIALE
li x4, 64	li x4, 64
loop	setvl x4
fld f1, 0(x1)	vld v1, x1
fld f2, 0(x2)	vld v2, x2
fadd.d f3, f1, f2	vadd v3, v1, v2
fstd f3, 0(x3)	vst v3, x3
addi x1, 8	
addi x2, 8	
subi x4, 1	
bnez x4, loop	

la load viene pagata una sola volta per tutto il vettore
= parallelismo

2:

LANE = singolo elemento in un vettore o insieme di alcuni elementi su cui si lavora in parallelo simultaneamente

Esempio: parallelismo a 8 lane \rightarrow si processano 8 elementi contemporaneamente + unità funzionale + porzione del VRF \rightarrow le functional unit lavorano su posizioni del register file. Se ad es. voglio caricare un vettore da 128 elementi in memoria gli elementi verranno inseriti nel RF col passo 4 \rightarrow Lane 0 contiene dati da 0 a 4, Lane 1 da 5 a 9.

vector length = lunghezza max in bit che l'R register può contenere

vettore
Se il vettore è grande il set-up time è trascurabile
se il vettore è corto si risente di + del set-up time

VECTOR CHAINING

E' l'analogo del forwarding

Ese. LV $v_2 \rightarrow$
MUL v_3, v_1, v_2

Nel devo aspettare che venga fatta la load di tutti gli elementi del vettore per iniziare a fare le moltiplicazioni (cosa che normalmente dovrei fare secca) (vector chaining).
questo secca passare dal RF

Se nelle LANE c'è sia l'unità fu che fa le load che le mull la vector chaining non funziona, quando ho alle streghe, cioè 2 istruzioni successive insieme sulla stessa unità fu. devo aspettare che si svuoti dal vettore poi riempire con v_2 .

START UP TIME: prima di vedere l'efficacia delle pipe, l'elemento deve scorrere attraverso tutti gli stadi. Sono i cicli che intercorrono da quando il 1° elemento entra nella pipe e quando esce.

DEAD TIME: tempo che serve a svuotare le pipe prima che un altro vettore la possa occupare.

SIMD



ARCHITETTURA AD ARRAY
(parallelismo spaziale)

l'unità fu processa in parallelismo
4 elementi dell'array

ARCHITETTURA VETTORIALE

pipeline

ogni u.f. lavora in sequenza sugli elementi dell'array.

STRIDE: distanza in memoria fra gli elementi consecutivi del vettore

stride = 1 \rightarrow gli elementi sono consecutivi nell'ordine in memoria

stride > 1 \rightarrow ci sono intervalli tra gli elementi

Unità di esecuzione vettoriale: fortemente pipeline, a ogni ciclo può prendere un nuovo elemento del vettore e farci operazioni.
Non ci sono mai dipendenze di dato.

2a

Se voglio nel mio acceleratore vettoriale non solo 1 unità fai ma ne voglio N, si mantengono le pipe ma se ne mettono N in //.

NOTA: le unità su delle istruzioni vettoriali sono diverse da quelle x le istruzioni scalari, la pipe è diversa, se non ci sono istruz. vett. ma solo scalari, la pipe delle vett. rimane vuota.

VECTOR INSTRUCTIONS SET:

- Compatto: una sola istruzione comanda N operazioni
- Espressivo: le N operazioni sono indipendenti, usano le stesse unità funz. accedendo alla memoria in sequenza (unit stride) o con un pattern conosciuto (strided)

AR. SIMD

↓
sfrutta parallelismo
microarchitetturale e
livello spaziale

ARCH. VETT.

↓
sfrutta pipelining, cioè
tutte le pipe processano elementi
diversi del vettore.
consentono anche repliche delle Mf.
per lavorare in //.

VPU ha registri di una dimensione fisso, ad es. 128 bit.

Se lavoro con struttura a 1024 bit, si conclude il calcolo sull'array con
8 iterazioni de 128.

Se l'array è lungo 1400 (mece fatto N iterat. a 128 poi una finale delle
quantiè rimanente).

↓
+ possibili operazioni a max //

Indirizzamento delle

LOAD / STORE

UNIT STRIDE

Indirizzamento lineare
con stride unitario
Vengono salvaguardati in
memoria elementi contigui
del vettore

NON-UNIT STRIDE

Stride non unitario,
stessa distanza fissa
tra il primo e il
secondo elemento
del vettore

INDEXED (GATHER-SCATTER)

Indirizzamento in cui
viene fornito un array
di indici da sommare
all'indirizzo base con
ai accedere in memoria

$$A[i] = B[i] + C[D[i]]$$

Si accede prima a $D[0]$ che
contiene un indice e con
quello indice si accede a C
per prendere gli elementi

questo codice fa inizialmente
il vettore B con gli elementi del
vettore B a partire da un registro
base del vettore D .

Accesso sparso in memoria
in cui esiste un vettore
di indirizzi

GATHER

legge da un vettore
deve e scrive in memoria
in modo non continuo

SCATTER

legge in modo
sparso dalla
memoria e dispone
in modo ordinato
in un vettore

?? (sì)

Se voglio vettorizzare un loop?

```
for (i=0; i<N; i++)  
    if (A[i] > 0) then  
        A[i] = B[i];
```

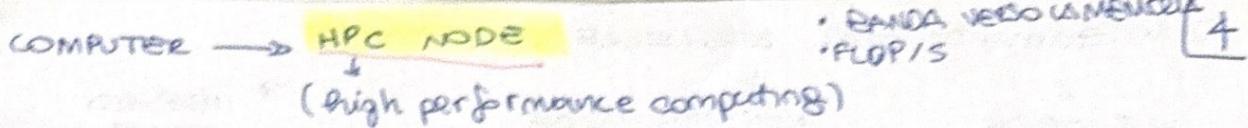
VECTOR CONDITIONAL EXECUTION

c'è dipendenza!
(esse non contemplate
gli 'one')

Soluzione: usare le maschere.

Nel vettore maschera ad ogni bit corrisponde un elemento del vettore dati, dove un bit impostato su 1 indica che l'elemento corrispondente deve essere incluso nell'operazione e un bit impostato su 0 indica che l'elemento deve essere escluso.
E quindi fare una selezione selettiva degli elementi all'uno eseguire operazioni.

SVE : scalable vector extension



performance di picco viene valutata in floating point al secondo
frequente e n° di core (//diffread)

FMA: fuse multiply add : prende 3 operandi sorgente e 1 destinat.
moltiplica due numeri e li somma con un terzo FMA f_1, f_2, f_3, f_4

$$f_a: f_2 + f_3 \times f_4 \rightarrow f_1$$

In un ciclo si fanno due operazioni floating point (questo solo x operazioni matriciali).

Tutti i processori hanno due frequenze : 1 frequenza nominale

Freq. di turbo: il processore puo' andare + veloce della freq. nominale e cio' dipende da parametri ambientali (raffreddamento, n° cores attivi...) tipicamente e 3,8 GHz. (n° cores · 3,8GHz = giga operaz / s)

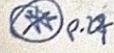
Se si ragiona su 1 sola istruzione: es. si hanno 52 core: $52 \cdot 3,8 = 190$

Nel calcolo dei FLOP/S si considera il caso migliore, ovvero che vengano fatte le FMA

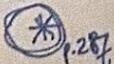
Giga operazioni/sec.

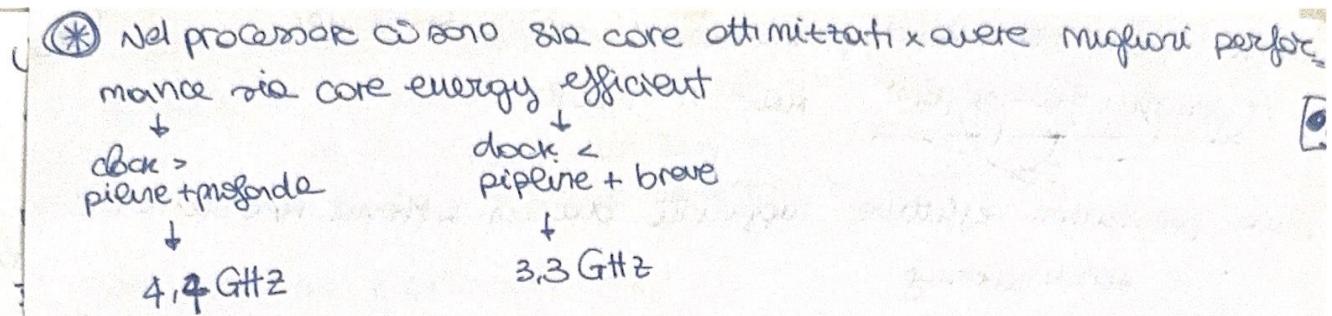
A volte si considerano le Flop/Watt x tenere in considerazione l'energia. 

• Top 500 → si basa sull'HPL algorithm

HPL (high performance linpack) : benchmark per supercomputer
e' basato sul risolvere un grande sistema di equazioni lineari dense
usando l'algoritmo di fattorizzazione LU di Gauss
HPL e' tipicamente usato x valutare le prestazioni computazionali
 p.17

STREAM: benchmark che serve x valutare la larghezza delle bande
delle memorie (GB/sec.)

lo stream si basa su 4 operazioni sulle memoria: scritture letterali
copie e scalare, eseguite su grandi array di dati e si
valuta lo throughput  p.27



(*) Variando la dimensione della matrice si rende il calcolo sempre + dipendente dalla capacità algebrica del processore rispetto al n° di dati che devono essere usati.

$\uparrow N$

$\uparrow C \uparrow N^3$

Facendo grande la matrice si valuta bene le capacità algebriche. HPL restituisce FLOP/S

(*) Alloca 3 array e fa diverse operazioni tra loro; fa la copia di un array all'altro, fa le somme di un array A con array B e copia i dati in C; fa le somme fra i due array moltiplicando per una costante un numero dei due array.

Restituisce una misura di MByte/s trasferiti.

RPEAK: picco teorico di prestazioni di un sistema HPC. Danno verso la memoria.

[6]

n° operazioni floating point \rightarrow max n° FLOPS

TCR

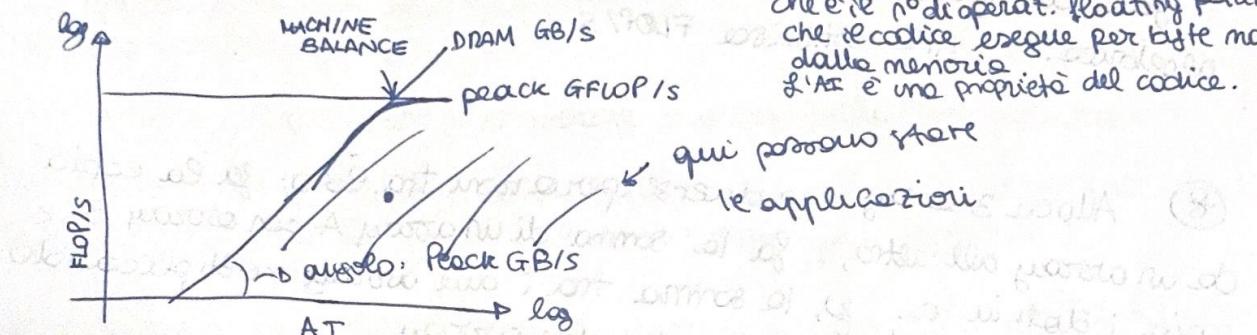
RMAX: prestazioni effettive raggiunte da un sistema HPC durante il benchmarking

Si mettono insieme FLOPs e bande.

ROOFLINE MODEL

Ci introdotte nel 2009, ricordate il tetto di una cosa) tempo di esecuzione di una generica operazione. Tex = n° istruz. CPI / fct / p. Modello concettuale usato per valutare e visualizzare le prestazioni di un'applicazione o di un algoritmo su un sistema di calcolo.

- Mette in ordine le performance computazionali e quelle della memoria relativamente a una macchina.



$$\text{GFLOP/s} = \min \left\{ \frac{\text{Peak GFLOP/s}}{\text{AI}}, \frac{\text{Peak GB/s}}{\log} \right\}$$

Machine balance \rightarrow tipi comuni 5-10 FLOPs per byte.

AI = arithmetic intensity : misura delle località dei dati
(riutilizzo dei dati)

L'assunzione che fa il modello realistico è che l'applicazione è composta da una fase di calcoli floating e una fase di spostamento dati da e verso la DRAM e assume che queste due fasi siano totalmente sovrapposte mentre si accede ai dati si fanno le operazioni floating point.

Il modello dice che l'execution time:

$$Time = \max \begin{cases} \# FLOPs / Peak GFLOP/s \\ \# Bytes / Peak GB/s \end{cases}$$

il tempo di esecuzione
è dominato dal più lento

↓
 velocità con cui leggo/scrivo byte dalla mem.
 velocità con cui si fanno operazioni floating point / sec.

$$\frac{Time}{\# FLOPs} = \max \begin{cases} 1 / Peak GFLOP/s \\ \# Bytes / (Peak GB/s \cdot FLOPs) \end{cases}$$

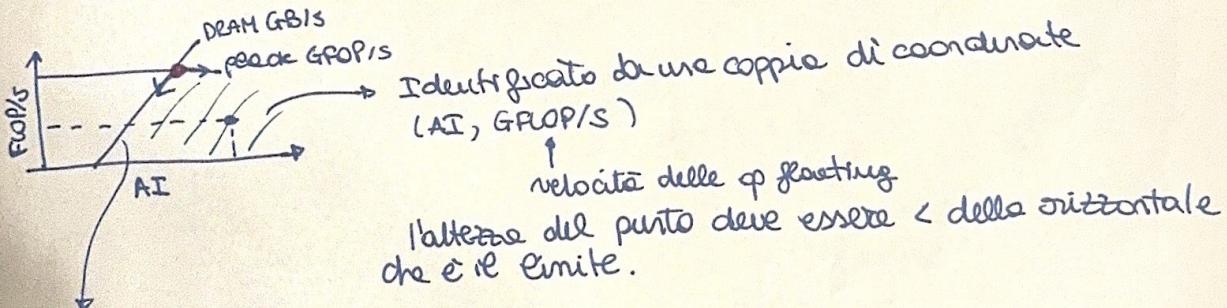
Time = min $\left\{ \frac{\text{Peak GFLOP/s}}{(\# FLOPs / \# Bytes) \cdot \text{Peak GB/s}} \right\}$

AI

$$GFLOP/s = \min \begin{cases} \text{Peak GFLOP/s} \quad (1) \\ AI \cdot \text{peak GB/s} \end{cases}$$

↓
ie n° di byte letti/scritti sono misurati rispetto alla DRAM

I GFLOP/s non possono essere maggiori del limite teorico (2)
è uguale all'AI · peak GB/s. Disegno questi limiti e ottengo il modello.



Il coefficiente angolare è la peak GB/s
questo linea è legata alle max velocità di trasferimento dati verso le memorie.

• punto di machine balance: sfruttamento migliore possibile del processore.

Se ci si sposta a sx siamo vincolati da DRAM GB/s se ci si sposta a dx se si è vincolati da peak GFLOP/s

$\approx 5 \text{ a } 10 \text{ flop/Byte}$

Tutte le applicazioni che arrivano vicine a uno dei due
neog sono applicazioni che riescono a saturare la capacità
computazionale dell'architettura.

Applic. che sbattono contro il limite spostamento dalla
memoria sono application che sfruttano la banda al 100%.

App. che sbattono contro il picco di GFLOPs sono ~~app.~~
app. che saturano la capacità computaz. del proc.

App. che non sbattono in nessun limite hanno problemi che
possono essere di:

- codice scritto male
- microarchitetture → (es. non si usa le FMA)
- dipendente di dato