Principali linee di evoluzione architetturale

Memoria cache e gerarchia di memoria

Principio di funzionamento della memoria cache

La memoria cache e' stata introdotta per ottimizzare le prestazioni della memoria di lavoro. Inizialmente la velocita' della memoria di lavoro DRAM e della CPU erano dello stesso ordine di grandezza, ma partire dagli anni 80 la frequenza di lavoro delle CPU Intel i86 crebbe molto velocemente. Verso la fine degli anni 90 il gap e' gia' di ordine 10. Come e' possibile che i PC funzionassero (e funzionino) con una memoria di lavoro 10 volte piu' lenta della CPU? Se ogni volta che la CPU deve esegure il fetch dobbiamo attendere 10 giri di clock per accedere a memoria c'e' qualcosa che non torna...

Se plottiamo il grafico tempo-indirizzi di memoria a cui la CPU accede, notiamo come, tranne in casi isolati, siano presenti delle "zone calde", ossia gruppi di celle di memoria non necessariamente contigue, ma comunque vicine che vengono interrogate in sequenza, seguendo dei "pattern regolari".

Questo fenomeno prende il nome di **principio di localita'** degli accessi a memoria da parte della CPU:

> Se all'istante t la CPU genera l'indirizzo di memoria xNNNN, e' molto probabile che nell'immediato futuro generi di nuovo lo stesso indirizzo xNNNN o indirizzi vicini (**locali**) all'indirizzo xNNNN.

Ci sono 2 motivazioni per cui questo fenomeno avviene:

- Localita' **spaziale**:
  - Il fetch delle istruzioni procede normalmente in celle consecutive;
  - I programmi sono organizzati in blocchi, moduli, procedure, e le variabili del singolo blocco sono memorizzate in spazi di memoria vicini;
- Localita' **temporale**:
  - L'essenza della programmazione sono i **cicli**, l'esistenza di gruppi di istruzioni scritte una volta ed eseguite molte volte dal calcolatore. Quindi le istruzioni e le variabili scritte nei cicli vengono reiterate molte volte.

Quando la CPU genera un indirizzo di memoria portiamo il contenuto della cella rischiesta e un certo numero di celle vicine **blocco** in una memoria (a lettura e scrittura):

- Piu' veloce della DRAM;
- Ovviamente piu' piccola perche piu' costosa da realizzare.

Questa memoria prende il nome di **cache**, in derivazione dalla parola francese

cache' (nascosto) perche' la sua esistenza non e' nota ne' al programmatore ne' alla CPU (la CPU genera indirizzi di memoria di lavoro, pensati per la DRAM). La cache serve a sfruttare statisticamente il fatto che di tutte le celle della memoria di lavoro, solo un sottoinsime e' di piu' probabile utilizzo da parte della CPU, quindi di fatto la sua presenza serve solo a velocizzare gli accessi a memoria.

Intel era solita far uscire ogni 4 anni una nuova generazione di processori. Durante l'arco di vita del processore, il numero di MIPS massimi tendeva a raddoppiare (miglioramento **tecnologico**). Mentre tra una generazione e l'altra vediamo una triplicazione del numero di transistor che genera complessita' (miglioramento **architetturale**).

I transistor in piu' a partire dall'82 vennero impiegati per realizzare una memoria cache **a bordo** del chip della CPU in modo che possa lavorare alla stessa frequenza di clock: nacque quindi la **cache L1** (di primo livello) di qualche KB. La frequenza del processore pero' e' cresciuta a tal punto che la differenze con la DRAM sono diventate sempre piu' evidenti. Venne introdotta la **cache L2** (di secondo livello), stavolta **esterna** al processore di qualche centinaio di KB, tipicamente di tipo SRAM. In casi limite viene impiegata anche una **cache L3** (di terzo livello), sempre esterna al processore, di qualche decina di MB.

Memoria cache e politica Tag Associative

La cache e' contenitore di una copia dei blocchi di celle di memoria di lavoro che circondano la cella richiesta dalla CPU in un dato accesso alla memoria. Il progetto della cache richiede una struttura generale:

- Memoria di lavoro divisa in blocchi di dimensione predefinita (ad esempio 16 celle o parole), ma che non puo' essere troppo elevata altrimenti incappiamo in rallentamenti eccessivi durante il processo di copia;
- Memoria cache divisa in blocchi di dimensione uguale a quelli della memoria di lavoro;
- Ogni volta che non e' presente una cella indirizzata dalla CPU nella memoria cache, si trasporta all'interno della cache l'interno blocco di memoria di lavoro che la contiene.

Abbiamo pero' 3 problemi da risolvere:

- Mapping dei blocchi da memoria di lavoro a memoria cache: ogni blocco di memoria di lavoro in quale o quali blocchi di memoria cache puo' essere copiato;
- Ricerca della parola richiesta dalla CPU: quando la CPU genera un indirizzo di memoria dobbiamo verificare se la parola richiesta si trova in cache:
  - In caso positivo si parla di **HIT**;
  - In caso negativo si parla di **MISS**, dunque la parola cercata deve essere prelevata dalla memoria di lavoro rallentando notevolmente

l'esecuzione;
- Sostituzione del blocco di memoria cache in caso di MISS: quali azioni si devono intraprendere per ottimizzare l'uso della cache (fare in modo che l'**HIT-RATIO** sia il piu' alto possibile).

Mapping dei blocchi da memoria di lavoro a memoria cache Vediamo dunque come funziona la politica **Tag Associative** e per farlo facciamo riferimento alla memoria di lavoro della CPU LC-2 costituita da 64K parole. Supponiamo di dimensionare il blocco trasferibile su memoria cache a 16 parole, questo fa si che l'intera memoria di lavoro sia divisa in 4096 blocchi da 16 parole ciascuno. Supponiamo poi di avere una memoria cache costituita da 128 blocchi da 16 parole ciascuno. Nella polita Tag Associative si crea una corrispondenza univoca tra blocchi di memoria di lavoro e blocchi di memoria cache: un blocco di memoria di lavoro puo' essere copiato in uno e un solo blocco di memoria cache. Poiche' il numero di blocchi della memoria di lavoro e' molto superiore, abbiamo un rapporto **many-to-one** in quanto piu' blocchi di memoria di lavoro sono connessi ad un singolo blocco di cache. Ogni blocco di memoria cache e' connesso ad un **gruppo** di blocchi di memoria di lavoro.

Possiamo etichettare il singolo blocco di memoria di lavoro, suddividere il suo indirizzo (da 0 a 4095) in 2 parti:

- Una che ci dice in che posizione del blocco di memoria cache si trova rispetto al gruppo;
- Una che ci dice di quale gruppo di memoria di lavoro fa parte.

In qualsiasi istante avremo in cache una copia di alcuni blocchi di memoria di lavoro e ci e' facile intuire a quale gruppo fa riferimento un determinato blocco di memoria cache. Per ricordarci invece quale blocco di memoria di lavoro (rispetto al gruppo) si trova in cache segnamo il numero del blocco di memoria di lavoro in una memoria di **tag**. Variando il contenuto della memoria di tag teniamo traccia di quale blocco dei vari gruppi sono istante per istante presenti nella cache.

Dunque nella politica Tag Associative si definisce a priori una corrispondenza univoca tra:

- gruppo di blocchi in memoria di lavoro;
- blocco di possibile destinazione della cache.

L'indirizzo generato dalla CPU (a 16bit) ha la seguente struttura (partendo dai bit meno significativi):

- **NP: Numero di parola nel blocco** - 4bit (2^4=16 parole per blocco di memoria di lavoro);
- **NG: Numero del gruppo di blocchi** - 7bit (2^7=128 gruppi di memoria di lavoro / blocchi di cache);
- **NB: Numero del blocco nel gruppo** - 5bit (2^5=32 blocchi per gruppo di memoria di lavoro).

Ricerca della parola richiesta dalla CPU e sostituzione del blocco di memoria cache in caso di MISS Sull'Address Bus, la CPU indirizza una cella concatenando NB & NG & NP. Per scoprire se NB e' presente in cache andiamo a leggere nella tabella dei tag in posizione NG se NB e' caricato:

- HIT: la CPU interagisce con la cella di memoria cache di indirizzo NG & NP;
- MISS:
  - Preleviamo da memoria di lavoro il blocco necessario di indirizzo NB & NG e lo copiamo nel blocco di cache di indirizzo NG;
  - Il tag associato al gruppo NG viene aggiornato con il valore NB;
  - La CPU ora puo' interagire con la memoria cache di indirizzo NG & NP (come nel caso di HIT).

La differenza e' in termini di tempo, perche' sono necessarie 2 operazioni preliminari nel caso di MISS.

Caratteristiche del Tag Associative:

- **Politica semplice**: il blocco richiesto dalla CPU puo' trovarsi solo in un blocco di cache
  - la scoperta di HIT/MISS e' rapida e priva di problemi (basta leggere il contenuto dell'unico blocco di cache, indicato dal tag NG, che puo' contenere il blocco di memoria di lavoro);
  - In caso di MISS il blocco richiesto puo' essere ricopiato in un'unica posizione;
- **Politica non ottimizzata**: ogni blocco di memoria cache **ottimizza localmente** l'accessibilita' ai blocchi di memoria di lavoro assegnati (solo il blocco di memoria di lavoro piu' recente e' tenuto in cache rispetto al suo gruppo)
  - Se due blocchi di memoria di lavoro facenti parte dello stesso gruppo sono piu' gettonati (anche e soprattutto rispetto altri blocchi di memoria di lavoro di altri gruppi), lo sfruttamento dei blocchi di memoria cache non e' uniforme.

Politiche Fully Associative e Set Associative

La politica Tag Associative e' semplice da realizzare ma non ottimizza l'uso della memoria cache: il problema principale sta nell'**allocazione fissa** fra i blocchi di memoria di lavoro e memoria cache.

Mapping della politica Fully Associative La politica Fully Associative prevede un accoppiamento libero: qualsiasi blocco di memoria di lavoro puo' essere copiato in qualsiasi blocco di cache. Dunque sparisce il concetto di gruppo, di riflesso il tag aumenta di dimensioni.

La struttura dell'indirizzo generato dalla CPU e' diversa (partendo dal bit meno significativo):

- **NP: Numero di parola nel blocco** - 4bit (2^4=16 parole per blocco di memoria di lavoro);
- **NBMdl: Numero del blocco in memoria di lavoro** - 12bit (2^12=4096 blocchi di memoria di lavoro).

Servono pero' ulteriori elementi circuitali di una certa complessita'.

Per poter verificare rapidamente se il blocco richiesto e' in cache:

- La memoria dei tag deve essere una memoria ad **accesso associativo**:
  - Devo far vedere alla memoria di tag a quale numero di blocco di memoria di lavoro sono interessato;
  - Ottenere in un tempo di accesso l'indirizzo della cella che lo contiene e quindi il blocco di cache nel quale e' copiato il blocco di memoria di lavoro che cerco (oppure una segnalazione di assenza).

Per poter decidere dove scrivere il blocco cercato se non e' presente in cache (MISS):

- Facciamo uso della politica **LRU** (Least Recently Used) ossia sovrascriviamo il blocco di cache "piu' vecchio" con il blocco di memoria che ci serve;
  - Per sapere quale e' il blocco LRU si associa ad ogni blocco di cache un **contatore a saturazione**:
    - Viene azzerato quando si accede al blocco associato;
    - Incrementato di 1 quando si accede ad un altro blocco;
  - Se questi contatori, come valore di fine scala hanno il numero di blocchi di cache (nel nostro caso 128), si e' sicuri di avere sempre un contatore saturo (che contiene 11..11), ossia un blocco che da almeno 128 volte non e' stato utilizzato.

Ricerca parola in cache con politica Fully Associative La CPU genera sull'Address Bus l'indirizzo composto dalla concatenazione NBMdl & NP. Il gestore di cache fa un accesso associativo alla memoria di tag specificando NBMdl come contenuto del registro associativo e ottiene come risultato il numero di blocco in memoria cache il blocco di memoria di lavoro cercato:

- HIT:
  - La CPU interagisce con la cella di memoria indicata dal numero di blocco di memoria cache & NP;
  - Aumentiamo di 1 tutti i contatori a saturazione;
  - Azzeriamo il contatore del blocco di cache indirizzato.
- MISS:
  - Identifichiamo in quale blocco di memoria cache effettuare la ricopiatura facendo un accesso associativo ai contatori a saturazione

cercando il primo di essi che contiene tutti uni (11..11) in modo da trovare il blocco LRU;
  - Effettuiamo la copiatura dal blocco di memoria (NBMdl dato dalla CPU in ingresso) di lavoro al blocco di cache LRU;
  - Scriviamo l'identificativo NBMdl nei tag per tenere traccia del contenuto del blocco di cache;
  - Consentiamo alla CPU di fare l'accesso alla memoria cache (da qui in poi come nel caso di HIT);
  - Aumentiamo di 1 tutti i contatori a saturazione;
  - Azzeriamo il contatore del blocco di cache indirizzato.

Caratteristiche Fully Associative:

- **Politica ottimizzata**: i blocchi presenti in cache sono sempre quelli che nel recente passato sono stati piu' richiesti dalla CPU
  - Otteniamo uno sfruttamento omogeneo dei blocchi di cache;
- **Politica complessa e costosa**: la ricerca del blocco richiesto implica il ricorso a memoria associativa per i tag
  - Per implementare la politica LRU serve prevedere l'uso dei contatori a saturazione che devono anche essi essere accessibili in modo asssociativo.

Politica Set Associative Compromessi:

- Un blocco di memoria di lavoro puo' essere copiato in un **insieme limitato (set)** di blocchi di cache:
  - Si parla di **n-way cache Set Associative**:
    - Con vari valori di n che concidono con il numero di blocchi cache dove puo' essere copiato un blocco di memoria di lavoro (un singolo blocco di memoria di lavoro puo' essere copiato in n blocchi cache).
- Semplifica la progettazione circuitale ma non e' perfettamente omogeneo come la politica Fully Associative.

Accesso in scrittura alla memoria cache

Se la CPU fa una scrittura in memoria cache, il dato in cache viene modificato rispetto al valore precedente.

Cosa ne facciamo della cella originale nella memoria di lavoro?

Politica **store thru**: si modifica il dato sia in cache sia in memoria:

- Politica semplice perche le informazioni in memoria e le loro copie in cache rimangono sempre congruenti;
- Non si hanno ulteriori complessita' HW;

- Politica non ottimizzata perche' gli accessi a memoria in scrittura non vengono velocizzati dalla presenza della cache;
- E' accettabile solo perche gli accessi in scrittura in memoria da parte della CPU sono in numero decisamente inferiore di quelli in lettura:
  - Ad esempio: per produrre un risultato servono piu' operandi (molteplici letture, una sola scrittura);
  - Tutte le acquisizioni di istruzioni macchina da memoria di lavoro sono tutte operazioni di lettura (fetch).

Politica **store in**: si modifica il dato soltanto in cache:

- Politica ottimizzata, in quanto anche gli accessi in scrittura vengono velocizzati se trovano la cella desiderata in cache;
- Politica complessa perche' tra cache e memoria di lavoro si crea **incongruenza**, la copia piu' aggiornata e' la copia in cache:
  - Si potrebbe riscrivere il blocco da cache a memoria di lavoro prima di eliminarlo ma se i due blocchi sono uguali si perde tempo;
  - Si introduce un **bit di modifica** M per ogni blocco di memoria cache che viene azzerato quando carichiamo un blocco da memoria di lavoro a cache e portato a 1 se si verifica una scrittura che indica essere piu' recente rispetto alla sua copia in memoria di lavoro. Quando dobbiamo sostituire un blocco di cache, se M=1 lo si riscrive in memoria di lavoro.

Gerarchia di memoria

- **Pochi registri GPR** all'interno della CPU destinati ai dati in elaborazione;
- **L1 cache** all'interno della CPU costituita da qualche KB di memoria estremamente veloce perche' delle stesse caratteristiche tecnologiche della CPU;
- **L2 cache** fuori dalla CPU da qualche centinaio di KB o MB per ridurre la distanza di prestazioni tra CPU e memoria di lavoro;
- **Memoria di lavoro** e' il supporto principale di memorizzazione delle informazioni: dati e programmi accessibili dalla CPU, da centinaia di MB o GB;
- **Memoria di massa a disco** con caratterische diverse da memorie elettroniche da decine e centinaia di GB;
- **Memoria di backup ottica o a nastro** consente di memorizzare a costi estremamente contenuti diverse centinaia di GB.

La memoria di massa a disco magnetico:

- Ha un tempo di accesso di qualche msec;
- Dimensioni dell'ordine di decine di GB;
- E' la tipica memoria di massa "online":

  - Ha tutti i programmi che pensiamo di usare quotidianamente;
  - Ha tutti i dati su cui operiamo quotidianamente.

La memoria di massa a disco ottico o a nastro:

- Ha un tempo di accesso dalle centinaia alle decine di msec;
- Dimensioni decisiamente piu' ampie a costo modesto;
- Memoria di massa "offline" con dati e programmi non necessariamente indispensabili nell'immediato o ad uso frequente:
  - Copie originali dei programmi;
  - Backup dei propri dati per evitare che i danneggiamenti della memoria online provochino perdite.

Rapporto tra memoria DRAM e a disco magnetico. La memoria di lavoro (con tutti i livelli cache soprastanti) e' una memoria:

- Di natura **elettronica**, con tempi di lavoro in linea con quelli della CPU;
- Ad **accesso casuale**, ogni cella richiesta e' accessibile nello stesso tempo (cache permettendo).

La memoria a disco magnetico e' una memoria:

- Di natura **meccanica**, con tempi di lavoro inaccettabili per la CPU;
- Ad **accesso sequenziale** (o **misto**): il tempo di accesso varia a seconda della posizione sul disco richiesta dalla CPU
  - Il disco e' organizzato in **tracce** (piste circolari concentriche), ognuna divisa in **settori**;
  - Per raggiungere un settore dobbiamo muovere la testina lettura/scrittura radialmente per raggiungere la traccia in cui si trova il settore, e attendiamo la rotazione del disco in modo che la testina cada sul settore richiesto dalla CPU.

Disk cache Facciamo in modo che una parte della memoria di lavoro sia destinata a contenere i settori di disco piu' recentemente richiesti. Si velocizza dunque l'esecuzione dei programmi che accedono spesso a file su disco. E' una tecnica poco usata in quanto il guadagno con una disk cache e' molto dipendente dal tipo di programma (e in generale piuttosto modesto).

Memoria virtuale Si fa vedere ad ogni programma destinato ad essere eseguito, l'intero spazio di indirizzamento della CPU come se fosse tutto a sua disposizione e tutto pieno di memoria fisica:

- Si chiama **memoria virtuale** perche' sembra esserci, ma non esiste, infatti viene vista da ogni programma perche' parte dello spazio di indirizzamento che il programma crede di avere a disposizione viene caricato in DRAM, il resto rimane su disco.

- In caso di memoria virtuale, la CPU quando esegue il programma genera **indirizzi virtuali** (ossia indirizzi che fanno riferimento ad un intero spazio di indirizzamento privato di ogni programma in esecuzione);
- In questo modo la quantita' di memoria fisica presente non limita le dimensioni del singolo programma, limita solo le prestazioni (quanta piu' memoria fisica c'e', tanto piu' grande e' la parte del programma che crede di avere una memoria virtuale completamente a sua disposizione, e ampia come lo spazio di indirizzamento, tanto maggiore e' la memoria che e' davvero presente in memoria fisica).
- La quantita' di memoria fisica presente non limita il numero di programmi in esecuizione, ma ancora, solo le prestazioni (se tanti programmi condividono poca memoria fisica, sara' spesso necessario fare operazini di trasferimento di parti dello spazio di indirizzamento da memoria DRAM a disco, limitando dunque le prestazioni del sistema).

Ci sono diversi problemi da risolvere per gestire la memoria virtuale:

- Traduzione da indirizzo virtuale generato dalla CPU a indirizzo reale:
  - Per riuscire a fare quest'operazione in tempi ragionevoli e' necessario introdurre un dispositivo HW dedicato **MMU - Memory Management Unit**;
- Ricerca in memoria di lavoro della cella indirizzata nello spazio virtuale (dunque scoprire se l'indirizzo virtuale trovato e' un indirizzo che contiene l'informazione desiderata):
  - Nel caso questo non sia vero, e' necessario definire le modalita' di gestione dei trasferimenti di blocchi di celle da disco a memoria di lavoro e viceversa (per portare in memoria reale la parte di memoria virtuale effettivamente necessaria alla CPU):
    * Paginazione;
    * Segmentazione;
    * Segmentazione paginata.

Non esiste la memoria ideale (economica e velocissima) pero' tramite la gestione a gerarchia di memoria possiamo avvicinarci molto a una memoria ideale. Con il principio di localita' degli accessi sfruttato a tutti i livelli riusciamo ad avere uno spazio di memoria del costo e delle dimensioni dei nastri/supporti di backup, piu' il costo dei dispositivi intermedi. La disposizione sara':

- Dati e programmi quotidiani sui dischi magnetici;
- Dati e programmi in esecuzione sulle memorie DRAM a semiconduttore;
- La parte calda sui vari livelli cache.

Con uno sforzo di progetto architetturale riusciamo ad ottenere una memoria grande come un armadio di nastri e veloce come la cache L1 a bordo della CPU.

Strutture pipeline

Motivazioni del ricorso alle strutture pipeline

La CPU passa la sua esistenza ad eseguire istruzioni macchina e ognuna di queste viene portata a termine in diverse fasi:

- Fetch;
- Decode;
- Execute;
- Write Back (Scrittura finale del risultato per esempio nei registri GPR o celle di memoria).

Le varie fasi non fanno uso di elementi elettronici comuni, per questo possiamo adottare una catena di montaggio **pipeline**, che non velocizza il completamento di un'istruzione (che in realta' viene in parte rallentata) ma aumenta il quantitativo di istruzioni completate per unita' di tempo MIPS.

Con la pipeline, dopo il transitorio di avviamento, il tasso di completamento istruzioni (**ICR**: Instruction Completion Rate) e':

> ICRcpupipeline = NumeroDiOperai x ICRcpunormale Il numero di operai e' indicato con **NS** (Number of stages).

In realta' le 4 fasi svolte per ogni istruzione non hanno tutte la stessa durata dunque si procede con il ritmo dell'operaio piu' lento:

> ICRcpupipeline < NumeroDiOperai x ICRcpunormale

Problemi di esecuzione programmi con CPU pipeline

Durante l'esecuzione di programmi da parte di una CPU pipeline, si incontrano situazioni che comportano il rallentamento delle attivita' (**dependencies**) e quindi la perdita di efficacia. Possiamo distinguere 3 tipi di dipendenze:

- **Control** dependency: il flusso di esecuzione sequenziale delle istruzioni si interrompe (tipicamente in situazioni di salto);
- **Data** dependency: due istruzioni vicine utilizzano lo stesso dato;
- **Resource** dependency: due o piu' istruzioni entrano in conflitto per una risorsa del sistema come l'accesso a memoria di lavoro;

Control dependency In realta' i salti sono una minoranza delle istruzioni. I salti incondizionati possono essere risolti a livello di FetchControlUnit che, acquisendo un'istruzione di salto incondizionato, puo' intraprendere autonomamente la decisione di saltare senza rimandare alla WriteBackControlUnit questo compito.

Per migliorare l'efficienza con i salti condizionati (che richiedono il coinvolgimento della ExecuteControlUnit) si puo' usare un approccio statistico:

- Se l'istruzione di salto si era gia' incontrata, ci si comporta come nel caso precedente;
- Se non si era mai incontrata questa istruzione:
  - Se il salto e' all'indietro (ciclo) provo a saltare;
  - Se e' un salto in avanti (costrutti if, then, else), provo a non saltare.

Questo approccio statistico si basa sulla **Branch Prediction Table** che si trova a bordo della CPU e costituita in parte da memoria associativa. Questa tabella contiene:

- **Branch address**: Indirizzo dell'istruzione di salto;
- **Branch destination**: Indirizzo della destinazione di salto (relativa all'istruzione);
- **Statistics**: Bit di statistica che servono a dire che cosa era successo in quella istruzione.

Quando la FetchControlUnit incontra un'istruzione di salto, consulta il Branch address. Se l'istruzione era gia' stata incontrata ci adeguiamo a Statistics (**Taken/NotTaken**), qunidi nel caso il salto fosse stato intrapreso la volta precedente, la FetchControlUnit salta al Branch desintation. Se, invece, l'istruzione non era gia' stata incontrata, la FetchControlUnit inserisce un nuovo record nella Branch Prediction Table e salta indietro/non salta in avanti. Quando finalmente l'ExecutionControlUnit risolve il salto, aggiorna la Branch Prediction Table, e nel caso in cui la FetchControlUnit ha "sbagliato scelta", blocca l'andamento del programma, svuota la pipeline e costringe la FetchControlUnit di ricominciare a svolgere il proprio compito dalla corretta istruzione.

Data dependency Una possibile soluzione del Data dependency e' la tecnica del **Data Forwarding**. Quando la ExecutionControlUnit ha completato il suo compito, e il dato che emette e' necessario per l'istruzione successiva, **inoltra** il dato a se stesso e alla WriteBackControlUnit (come di norma).

Resource Dependency La tipica soluzione per i problemi di accesso a memoria e' la cosiddetta **Macchina di Harvard**. L'intera CPU presenta al proprio interno 2 strutture di Bus per accesso a due memorie cache L1 a bordo. Una destinata a contenere istruzioni e l'altra a contenere variabili, dati, informazioni.

In questo modo solo sul bus esterno che collega la cache L2 (se c'e', altrimenti memoria di lavoro) alla CPU possiamo incontrare conflitti, che comunque sono gia' risolti perche' siamo gia' in una situazione di MISS che ci costringe ad attendere un tempo maggiore.

La Macchina di Harvard ha spazi di indirizzamento separati per istruzioni e dati variabili, realizzata con due cache L1 a bordo della CPU, di fatto elimina i conflitti tra fasi di fetch e di accesso ai dati variabili, senza duplicare il System

Bus (quindi avendo un'unica memoria di lavoro collegata mediante il bus di sistema alla CPU).

Rimane solo un possibile conflitto tra ExecutionControlUnit e WriteBackControlUnit in caso di lettura e scrittura simultanea di dati variabili. Per evitare questi problemi si ricorre a tecniche di compilazione ottimizzata: dunque durante la fase di traduzione da linguaggio ad alto livello a istruzioni macchina, le istruzioni di LOAD e STORE vengono spostate in modo che non entrino in conflitto sull'accesso al **Variables Bus**.