**DOMANDE SECONDA PARTE**

**Spiegare la differenza tra la codifica dei numeri interi in complemento a due rispetto a quella in modulo e segno**

Nella rappresentazione in modulo e segno il bit più significativo (più a sinistra) rappresenta il segno, cioè ‘+’ se è 1 o ‘–‘ se è 0. I restanti numeri rappresentano il modulo. In questa rappresentazione però lo 0 ha due rappresentazioni (0000= +0 / 1000= -0), rappresentando un grande svantaggio nei calcoli.

Nella rappresentazione in complemento a 2 il bit più significativo resta il segno e i numeri rappresentabili con n bit vanno da -2n-1 a 2n-1. Nei numeri positivi il primo bit è 0 e il resto in forma binaria, nei numeri negativi si fa il complemento bit a bit del numero positivo corrispondente e si somma 1 al risultato.

Esempio: 20 = 00010100

-20 = 11101011+

1=

----------------

11101100

**Si spieghi in dettaglio la codifica in complemento a 2 dei numeri interi. Si discutano poi i problemi legati alla realizzazione della moltiplicazione di due interi rappresentati in complemento a 2, esemplificando tali problemi su un caso concreto di moltiplicazione**

La rappresentazione in complemento a 2 il bit più significativo rappresenta il segno, positivo se 0 e negativo se 1, e permette di rappresentare da -2n-1 a 2n-1 . I numeri positivi vengono rappresentati con 0 al primo bit e il resto in forma binaria, per numeri negativi si fa il complemento a 2 del rispettivo numero positivo sommandovi poi 1.

In una moltiplicazione con complemento a due il problema che può sorgere è la moltiplicazione del segno, andando a rendere errato il risultato se almeno uno dei due numeri è negativo. Per evitare questo si utilizza la rappresentazione in complemento a due per i prodotti parziali (algoritmo di Booth).

**Si spieghi in dettaglio l’hardware adottato per la realizzazione della somma e sottrazioni di numeri interi rappresentati in complemento a 2**

L’elemento che realizza la somma e la sottrazione è l’adder, un sommatore binario, al quale vengono forniti gli addendi/sottraendi e restituisce la somma/sottrazione di essi oppure un overflow, che genera un flag e viene posizionato in un apposito registro. L’overflow avviene quando il risultato della somma supera il numero di bit disponibili per il risultato oppure quando la somma di due numeri porta a un numero del segno opposto.

Nella somma i due addendi vengono da due registri e il risultato può essere salvato in uno dei due registri oppure su un altro registro. Nella sottrazione al sottraendo viene applicato il complemento a 2, sommando al minuendo l’opposto del sottraendo.

**Si spieghi in dettaglio la rappresentazione dei numeri reali secondo lo standard IEEE 754**

Lo standard IEEE 754 definisce la più importante rappresentazione in virgola mobile, è stato sviluppato per facilitare la portabilità dei programmi e per aumentarne la sofisticatezza.

Esso definisce un formato singolo a 32 bit e un formato doppio a 64 bit, rispettivamente con esponenti di 8 e 11 bit e mantisse da 23 e 52 bit. Il primo bit, come negli altri formati rappresenta il segno, la base è 2 e l’1 è implicito a sinistra della virgola.

Il formato doppio viene usato per i calcoli intermedi, avendo precisione doppia infatti può evitare possibili errori di arrotondamento e diminuiscono la possibilità di overflow.

Gli esponenti vengono normalizzati, cioè vi viene sommato -127 nel formato singolo e -1023 nel formato doppio.

1.mantissa normalizzata \* 2 esponente-127/-1023

Per normalizzare la mantissa si sposta a dx o a sx in modo da avere un solo 1 a sx della virgola.

In base alla direzione dello spostamento si definisce il segno dell’esponente, in base al numero di bit spostati si definisce l’esponente, al quale poi verrà tolto 127 (singola) o 1023 (doppia).

**Si spieghi in dettaglio la divisione fra numeri reali nello standard IEEE 754**

Innanzitutto, si controlla la presenza dello 0 al divisore, se è così si manda un segnale di errore. Poi l’esponente del divisore viene sottratto all’esponente del dividendo, togliendo la polarizzazione, che deve essere sommata. Vengono eseguiti dei controlli sull’overflow o sull’underflow, poi si esegue la divisione, si normalizza il risultato e lo si arrotonda.

**Si spieghi in dettaglio la moltiplicazione fra numeri reali nello standard IEEE 754**

Innanzitutto, si controlla la presenza dello 0, se uno dei due operandi è 0 si pone il risultato a 0.

Poi si esegue la somma degli esponenti e la sottrazione ad essi della polarizzazione, tenendo conto del possibile underflow o overflow. Si esegue la moltiplicazione, si normalizza il risultato e lo si arrotonda (la normalizzazione può portare a un overflow dell’esponente).

**Quali sono i fattori che determinano la lunghezza delle istruzioni di una CPU?**

Ciò che influenza e viene influenzato dalla lunghezza delle istruzioni è: la dimensione della memoria e la sua organizzazione, la struttura del bus, la complessità e la velocità della CPU. Si tende a cercare un compromesso tra necessità di avere un ampio set di istruzioni e necessità di risparmiare spazio. La lunghezza delle istruzioni inoltre dovrebbe essere uguale o multipla della dimensione del bus di memoria e dovrebbe essere anche multipla della lunghezza di un char, quindi di 8 bit.

**Si discuta in dettaglio in cosa consiste il formato variabile per le istruzioni. Se possibile dare esempi di formati variabili.**

Il formato di un’istruzione definisce la disposizione dei bit e deve includere un codice operativo e, in modo implicito o esplicito, zero o più operandi.

Il formato variabile delle istruzioni permette di avere una maggiore flessibilità nell’indirizzamento, con molte combinazioni possibili di riferimenti a registri e memoria e più modi di indirizzamento, a discapito della semplicità del processore.

Un esempio di formato variabile può essere il PDP-11, progettato per un linguaggio macchina potente e flessibile che rispettasse i requisiti di un microcomputer a 16 bit. Dispone di 8 registri generici a 16 bit, gode dell’indipendenza definita ortogonalità, i formati comprendono istruzioni a 0, 1 o 2 indirizzi, la lunghezza del codice operativo varia dai 4 ai 16 bit e i riferimenti ai registri sono lunghi 6 bit. Le istruzioni sono lunghe una parola (16 bit) o multipli (32 e 48 bit).

Un altro esempio è il formato VAX, progettato adottando due criteri: tutte le istruzioni dovrebbero avere un numero naturale di operandi e quest’ultimi devono presentare la stessa generalità nelle specifiche. Il risultato consiste in un codice operativo di 1 o 2 byte seguito da alcuni specificatori di operando a seconda dell’opcode. La lunghezza minima di istruzione è di 1 byte, ma si possono costruire istruzioni anche da 37 byte. Il numero di specificatori di operando può arrivare a 6. Uno di essi occupa 1 byte in cui i 4 bit più a sinistra specificano il modo di indirizzamento. Uno specificatore di operando spesso consiste in un solo byte, dove i i 4 bit più a destra specificano uno dei 16 registri generici.

**Si descrivano i possibili formati di codifica di un’istruzione, specificando per ogni formato la sua composizione tipica, i pregi e i difetti**

Ci sono tre tipi possibili di formati di codifica delle istruzioni. Il primo prevede le istruzioni con lunghezza fissa, avendo più formati cambiando i campi. Particolarmente efficiente in caso di pipeline e il caricamento delle istruzioni avviene in modo più veloce rispetto agli altri formati, ma manca di operandi indipendenti dall’opcode, quindi ne sono disponibili meno da utilizzare nelle istruzioni, mancando quindi di flessibilità.

Il secondo invece prevede istruzioni di lunghezza variabile, che dipende cioè dal numero di operandi specificati nell’opcode. Il terzo infine prevede più formati con differenti lunghezze fisse.   
Gli ultimi due formati permettono una maggiore flessibilità rispetto al primo formato descritto, ma incrementano di molto la complessità della CPU.

Le caratteristiche del formato a lunghezza fissa si avvicinano alla filosofia RISC, mentre quelle del secondo e del terzo formato si avvicinano alla filosofia CISC.

**Si descriva in dettaglio la modalità di indirizzamento indiretto, discuterne pregi e difetti. L’adozione di tale indirizzamento è favorita o sfavorita in un’architettura RISC? Motivare la risposta**

L’indirizzamento indiretto prevede che nel campo indirizzo dell’istruzione vi sia l’indirizzo di una cella di memoria, chiamata puntatore, che punta all’indirizzo effettivo dell’operando per l’istruzione. Questo modo di indirizzamento permette di indirizzare un gran numero di operandi, infatti parole di lunghezza n permettono di indirizzare 2n entità diverse, ma ha come svantaggio un doppio accesso in memoria, quello per il puntatore e quello per l’operando. L’indirizzamento indiretto è sfavorito in un’architettura RISC, in quanto in essa si ottimizza l’accesso alle variabili locali aumentando il numero dei registri e si utilizza un set di istruzioni semplificato proprio per minimizzare gli accessi in memoria.

**Si descrivano in dettaglio le modalità di indirizzamento con spiazzamento e a pila. In particolare, si confrontino criticamente i due modi di indirizzamento e se ne discutano pregi e difetti**

L’indirizzamento con spiazzamento è un metodo che combina indirizzamento indiretto e registro indiretto. Il campo indirizzo infatti è suddiviso in due sottocampi, l’indirizzo A, che è il valore di base, e il registro R, che contiene il valore da sommare ad A per ottenere l’indirizzo situato in memoria. La pila invece è una sequenza lineare di locazioni riservate della memoria, vi è associato un puntatore SP contenuto nel registro e contiene l’indirizzo della cima della pila. Nell’indirizzamento a pila l’operando è il primo della fila, proprio come nel caso di un indirizzamento a registro diretto.

**Descrivere i possibili approcci per trattare l’indirizzo di ritorno da una chiamata di procedura**

La procedura è un pezzo di programma a cui si da un nome, in modo da poterlo chiamare ed eseguire da qualunque punto del programma, questo evita di scrivere più volte lo stesso pezzo di codice. La procedura può essere invocata tramite l’istruzione di chiamata e, grazie all’istruzione di ritorno (salto), una volta eseguita la procedura si ritorna al punto in cui si è interrotto il programma. Per trattare il ritorno da una procedura esistono 3 modi efficaci: Usare un registro, memorizzare l’indirizzo a inizio procedura o usare la cima di una pila. Nel primo modo ogni volta che avviene una chiamata a procedura si provoca il salvataggio in RN del PC e della lunghezza dell’istruzione, cosi da poter usare quei dati con la chiamata di ritorno. Nel secondo modo la procedura salva PC e la lunghezza dell’istruzione all’interno di sé stessa, ciò rende questo modo pratico e sicuro. Nell’ultimo modo invece gli indirizzi di ritorno vengono memorizzati in cima alla pila e vengono presi nell’ordine inverso alla chiusura delle procedure, in modo da gestire efficacemente anche chiamate di procedura annidate.

**Nel contesto di una pipeline descrivere la problematica della dipendenza da controllo e si discuta in particolare la tecnica del buffer circolare spiegando in quali situazioni tale tecnica è particolarmente efficace**

La dipendenza da controllo si verifica quando ci sono istruzioni che alterano la sequenzialità, come ad esempio i salti. Tra le diverse soluzioni c’è il mettere in stallo la pipeline finché non si è calcolato l’indirizzo della prossima istruzione (inefficiente) oppure individuare le istruzioni critiche per anticiparne l’esecuzione mediante l’adozione di un metodo di controllo. La soluzione del buffer circolare per i salti condizionati consiste nell’utilizzare questa memoria piccola e veloce formata da registri che mantengono le ultime n istruzioni prelevate. In caso di salto si controlla se la destinazione è già nel buffer. Tra i vantaggi troviamo la possibilità di anticipare il fetch di determinate istruzioni presenti nel BR e l’utilità in caso di loop dove tutte le istruzioni sono già presenti nel BR. Uno svantaggio è che il BR, essendo una memora piccola, può contenere un numero limitato di istruzioni, quindi per salti di molte istruzioni è necessario l’accesso in memoria.

**Nel contesto di una pipeline descrivere nel dettaglio la tecnica del data forwarding: a cosa serve?**

Il data forwarding è una delle possibili soluzioni alla dipendenza dai dati in una pipeline. C’è la possibilità infatti che una certa fase non possa essere eseguita in un determinato ciclo di clock perché i dati non sono ancora pronti, ed è qui che entra in atto il data forwarding. Questa tecnica sfrutta il fatto che il dato si conosca prima che venga aggiornato in memoria, quindi è possibile intercettarlo e propagarlo in avanti diminuendo i cicli di stallo. Grazie a questa tecnica è possibile passare il dato tra fasi EX-EX o tra MEM-EX.

**Nel contesto di una pipeline si consideri la tecnica che utilizza la tabella della storia dei salti. A cosa serve? Descriverla in dettaglio**

La tabella della storia dei salti è una piccola memoria cache associata alla fase IF. Ogni riga è costituita da 3 elementi: indirizzo di istruzione di salto, un certo numero di bit di storia e informazioni inerenti all’istruzione. Ogni prefetch innesca una ricerca all’interno della tabella, se non viene trovata una corrispondenza si utilizza per il prelievo il successivo indirizzo di sequenza, se invece si riscontra una corrispondenza si effettua una previsione sulla base dello stato dell’istruzione. Quando l’istruzione di salto viene eseguita lo stadio di esecuzione segnala il risultato alla tabella della storia dei salti. Lo stato dell’istruzione viene aggiornato per riflettere una previsione corretta o errata. Quando invece si incontra un’istruzione di salto condizionato non presente nella tabella, quest’ultima viene aggiornata cancellando una riga esistente scelta in base a uno degli algoritmi di sostituzione in vigore nella cache.

**Nel contesto di una pipeline descrivere nel dettaglio la tecnica di predizione del salto utilizzando 2 bit di predizione**

Per anticipare un salto possono essere utilizzati vari approcci dinamici di predizione che cercano di migliorarne la qualità memorizzando la storia delle istruzioni di salto di uno specifico programma. In particolare, posso associare un bit a ogni istruzione di salto condizionato che ne riflettano la storia precedente (bit taken/ not taken switch). Essi spingono il processore a prendere una determinata decisione all’occorrenza dell’istruzione. Utilizzando 2 bit di predizione è possibile memorizzare il risultato degli ultimi due salti. L’algoritmo legge la prossima istruzione di salto condizionato, finché le successive istruzioni di salto condizionato vengono effettuate si prevede che il prossimo salto verrà intrapreso. Se c’è un singolo errore di previsione l’algoritmo continua a prevedere che il salto verrà effettuato, in pratica richiede che due previsioni siano errate prima di cambiare la propria decisione di predizione (inverte il bit solo se ci sono due previsioni errate di seguito).

**Nel contesto di una pipeline si spieghi in dettaglio la tecnica del salto ritardato. Fornire un esempio**

Il salto ritardato è una delle soluzioni possibili alla dipendenza dal controllo. In pratica è un modo per incrementare l’efficienza della pipeline e fa uso di un salto che non ha effetto fino al termine dell’istruzione seguente. Finché non si sa se ci sarà o no il salto infatti si può eseguire un’istruzione che non dipende dal salto. La CPU esegue sempre questa istruzione e solo dopo altera la sequenza delle esecuzioni delle istruzioni. Nell’istruzione successiva al salto (branch delay slot) il compilatore cerca di inserirvi un’istruzione indipendente e non dannosa per il salto. Questo scambio funziona con successo nei salti incondizionati e nelle chiamate a procedura, ma non va usato ciecamente in caso di salto condizionato, perché la condizione da verificare può essere alterata a causa dello scambio di istruzioni fatto dal compilatore.

**Nella pipeline MIPS spiegare come lo stadio ID (instruction Decode) è in grado di rilevare la dipendenza fra i dati**

Nella fase ID è possibile rilevare tutte le dipendenze dai dati, dato che in questa fase si vanno a prelevare i dati che richiede l’istruzione. Ogni volta che si rileva una dipendenza dai dati l’istruzione va in stallo nella fase ID ed è possibile da qui determinare il tipo di data forwarding da adottare e predisporre gli opportuni segnali di controllo. Ci sono molti casi da considerare per effettuare il data forwarding, ma l’osservazione principale è che i dati provengono dall’output della ALU e della memoria dati (fase EX e MEM) e sono diretti verso l’input della ALU e della memoria dati (fase EX e MEM), quindi occorre confrontare i registri destinazione di IR in EX/MEM E MEM/WB con i registri sorgente di IR in ID/EX e EX/MEM.

**Nel contesto della pipeline descrivere la problematica della dipendenza dei dati e si discutano in dettaglio le tecniche viste a lezione per trattare il problema**

Un Data Hazard si verifica quando c’è un conflitto nell’accesso alla locazione di un dato. Ci sono tre possibili tipi di conflitto: RAW (read after write) o dipendenza effettiva, quando un’istruzione modifica un registro o una locazione di memoria e un’istruzione successiva legge il dato in quella locazione, WAR (write after read) o antidipendenza, quando un’istruzione legge un registro o una locazione di memoria e un’istruzione successiva scrive nella stessa posizione, WAW (write after write) o dipendenza di output, quando due istruzioni devono scrivere nella stessa locazione. Come possibili soluzioni si possono introdurre fasi di nop (non operative) per ritardare l’istruzione che causa il Data Hazard, si può utilizzare il metodo di data forwarding, che permette di passare in avanti il dato direttamente dall’uscita della ALU, si possono riordinare le istruzioni in modo da ridurre i Data Hazard oppure risolvere tramite MIPS a livello del compilatore.

**Spiegare nel dettaglio la differenza fra un’architettura CISC e una RISC**

L’architettura RISC è caratterizzata da un ampio numero di registri e da un set di istruzioni semplice e limitato, ciò grazie all’utilizzo di linguaggi ad alto livello. L’architettura CISC invece prevede l’utilizzo di un set di istruzioni ampio e complesso, il quale dovrebbe semplificare il compilatore e migliorarne le performances. Nella prima architettura ogni istruzione necessita di un ciclo di clock, nella seconda invece più cicli di clock per completare un’istruzione, e mentre nelle RISC le operazioni siano principalmente da registro a registro tranne LOAD e STORE nelle CISC ci sono anche operazioni memoria a memoria e registro a memoria. L’unità di controllo delle RISC è cablata, ciò la rende meno flessibile ma più veloce, Nelle CISC invece è microprogrammata, il che la rende più flessibile ma meno veloce. La RISC inoltre presenta pochi e semplici modi di indirizzamento, oltre a pochi formati fissi per le istruzioni, con opcode e campi a dimensione fissa.

Non è evidente quale sia l’architettura migliore, infatti spesso molte CPU presentano caratteristiche prese da entrambe le architetture, come ad esempio il Pentium II, che è CISC con elementi RISC.

**Spiegare le caratteristiche base dell’architettura RISC**

Le caratteristiche comuni a tutte le architetture RISC sono principalmente 4: la prima prevede il completamento di un’istruzione in un ciclo di clock, così da togliere la necessità di utilizzare microcodice; la seconda prevede operazioni da registro a registro, ciò implica che tutti gli operandi siano salvati nei registri ( per questo che nelle architetture RISC è presente un ampio numero di registri ), e per accedere alla memoria bastano LOAD e STORE, riducendo inoltre la complessità delle istruzioni e della CPU; la terza prevede pochi e semplici modi di indirizzamento, infatti quasi tutte le istruzioni RISC utilizzano l’indirizzamento a registro, sempre con il fine di semplificare le istruzioni e la CPU, inoltre altri modi di indirizzamento possono essere sintetizzati via software; la quarta caratteristica infine prevede dei formati semplici per le istruzioni, esse infatti hanno una lunghezza fissa e sono allineate sui confini di parole, inoltre le posizioni dei campi sono di dimensione fissa, in particolare gli opcode a dimensione fissa portano al vantaggio di poterne effettuare la decodifica simultaneamente, quindi viene ottimizzata la fase di fetch.

**Spiegare le motivazioni alla base dell’architettura CISC**

Con l’evoluzione dei calcolatori si riscontra un costo maggiore di software rispetto al costo dell’hardware, inoltre i linguaggi ad alto livello stavano diventando sempre più potenti e complessi. Ciò porta ad un buco tra ciò che era consentito fare grazie ai linguaggi ad alto livello e quello che effettivamente poteva essere eseguito dall’architettura del calcolatore. Le architetture CISC nascono proprio per ridurre questo “gap semantico” presentando una CPU microprogrammata che, a differenza di quella cablata, permetteva di poter effettuare più operazioni, migliorandone la flessibilità a discapito della velocità. Inoltre, presentavano un ampio set di istruzioni, il che permetteva di supportare linguaggi ad alto livello più complessi e, allo stesso tempo, facilitando la scrittura del compilatore. In teoria con l’introduzione delle architetture CISC si sarebbero ottenuti programmi più piccoli e veloci composti da un minor numero di istruzioni, anche se in realtà non è sicuro che sia più piccolo in termini di dimensioni, e la velocità non è assicurata data la complessità della CPU microprogrammata.

**Si metta a confronto criticamente il modo in cui un’architettura RISC utilizza l’ampio banco di registri a sua disposizione rispetto alla gestione di una cache**

Nelle architetture RISC il banco di registri, organizzato in finestre, agisce come un buffer circolare conservando le variabili che hanno più probabilità di essere utilizzate più frequentemente. Questo tipo di organizzazione assomiglia a quello di una memoria cache, anche se in realtà presentano diverse differenze. Innanzitutto, il banco di registri contiene le N-1 variabili scalari delle procedure attivate più recentemente, mentre la cache contiene una selezione di variabili scalari usate di recente. La cache gestisce lo spazio in modo più efficiente operando dinamicamente, inoltre tratta tutti i riferimenti alla memoria allo stesso modo, risulta però più lenta di un banco di registri, il quale gestisce inefficientemente lo spazio dato che non tutte le procedure richiedono l’intera dimensione della finestra. Infine, mentre la cache è in grado di trattare variabili locali e globali, nel banco di registri il trasferimento tra registri e memoria è determinato dalla profondità di annidamento della procedura.

**Spiegare in dettaglio come le architetture RISC trattino efficacemente le chiamate annidate di procedura**

In un’architettura RISC le chiamate annidate di solito non coinvolgono più di 6 parametri, ciò porta a un grado di annidamento non elevato, si è pensato quindi di suddividere i registri in piccoli gruppi di taglia fissa, detti finestre di registri. In questo modo ogni volta che c’è una chiamata si cambia automaticamente il gruppo di registri senza dover salvare i dati in memoria, e al ritorno viene selezionato il gruppo assegnato in precedenza. Ogni finestra è suddivisa in tre sottogruppi: parametri passati alla procedura, registri che salvano il contenuto delle variabili locali e registri temporanei che vengono usati per scambiare i parametri con la procedura chiamata. I registri dei parametri e i registri temporanei sono fisicamente gli stessi, questo perché i primi si sovrappongono perfettamente con i secondi, ciò permette il passaggio dei parametri senza trasferimento dei dati.

La realizzazione fisica delle finestre di registri avviene tramite buffer circolare. In esso quando una procedura ne chiama un’altra il puntatore CWP (current window pointer) avanza fino alla fine del registro locale della finestra attiva. Quando si esaurisce lo spazio nel BR, cioè quando i due puntatori, CWP e SWP (saved window pointer), si sovrappongono, si genera un interrupt dove la finestra più vecchia viene salvata in memoria e SWP, che indica dove ripristinare l’ultima finestra salvata in memoria, viene incrementato. Ogni volta che termina una procedura CWP viene decrementato di una finestra. Quando CWP=SWP di nuovo si genera un interrupt dove, se necessario, vengono ripristinate le procedure precedentemente salvate in memoria e SWP viene decrementato.

**Spiegare in quale modo un compilatore possa aiutare l’utilizzo efficace dei registri da parte di un’architettura RISC**

L’ottimizzazione dei registri in un’architettura RISC viene gestita dal compilatore. Esso associa ogni variabile a un registro simbolico che può contenere più variabili. Poi mappa ogni registro simbolico a un registro reale, che ne può contenere più di uno, in questo modo se due registri simbolici si usano in momenti diversi possono essere mappati sullo stesso registro reale. I registri reali però sono fisicamente finiti, al contrario dei registri simbolici che, essendo virtuali, possono essere infiniti, quindi quando i registri reali non sono in numero sufficiente per contenere le variabili riferite in quell’intervallo, alcune variabili vengono mantenute in memoria principale.

**Nel contesto MIPS spiegare come e quando la CPU attiva i circuiti di bypass**

La CPU attiva i circuiti di bypass nel caso in cui si necessiti di prelevare un valore che è stato calcolato dalla ALU e richiesto in un input da istruzioni successive. In questo caso si attiva **EX.ALUOutput\_istr1 🡪 EX:Top(o Bottom)\_ALU\_input\_istr2**, dove il dato passa dalla fase EX precedente alla fase EX successiva. Ci sono casi in cui il dato è disponibile all’uscita della fase MEM, e anche in questi casi si può attivare il bypass per prelevare il dato dall’uscita della fase MEM per portarlo in input alla ALU in fase EX di un’istruzione successiva, attivando

**MEM.LMD\_lw 🡪EX.Top\_ALU\_input\_istr**. Questo caso capita quando il dato da utilizzare viene precedentemente prelevato da un’istruzione di LOAD dalla memoria.

**Si descrivano i formati delle istruzioni MIPS visti a lezione specificando per ogni formato la sua composizione tipica**

Nel contesto MIPS ci sono tre formati di istruzioni da 32 bit: il formato R (Registro) viene utilizzato per le operazioni aritmetiche e logiche, ed è formato dai campi opcode (6 bit) che identifica l’operazione, rs (5 bit) che identifica il primo argomento, rt (5 bit) che identifica il secondo argomento, rd (5 bit) che identifica il registro destinazione, shamt (5 bit) che nelle operazioni di shift dice di quanto fare lo shift e funct (6 bit) che identifica la variante operativa; il formato I (Immediato) viene utilizzato per le operazioni di LOAD/STORE, istruzioni immediate e di salto condizionato ed è formato dai campi opcode (6 bit) che identifica l’operatore, rs ( 5 bit) che ientifica il primo argomento, rt (5 bit) che identifica il secondo argomento e address/const (16 bit) che indica il bit di spiazzamento da sommare al valore contenuto nel registro; Il formato J (salto) viene utilizzato per le istruzioni di salto incondizionato ed è formato dai campi opcode (6 bit) che identifica l’operazione e address (26 bit) che individua l’indirizzo della prossima istruzione da eseguire.

**Si descriva sinteticamente l’implementazione delle istruzioni attraverso la tecnica della microprogrammazione. Si dica se questa tecnica viene utilizzata per i processori RISC o CISC, motivare la risposta**

Nella microrprogrammazione la PC della CPU realizza il flusso di controllo appropriato per ogni istruzione inviando segnali alla parte operativa. Questi segnali vengono detti microistruzioni, e possono essere di trasferimento dati tra registri, di trasferimento dati tra registri e unità esterna, trasferimento dati da unità esterna a registro o di esecuzione di una operazione che usi i registri come I/O. I compiti alla base della PC sono di serializzazione ( determinare cioè la sequenza di microoperazioni da eseguire in funzione del codice operativo dell’istruzione) e di esecuzione ( cioè di far eseguire le microoperazioni), questo grazie alla generazione di segnali di controllo. La PC può essere cablata, come in una RISC, o microprogrammata, come in una CISC. Quest’ultima, con livello di astrazione 1, permette una maggiore flessibilità in fase di progettazione al contrario della prima che è realizzata tramite circuiti cablati e ha quindi un livello di astrazione 0.

A livello di firmware il microprogramma di un’unità riunisce i frammenti di programma delle diverse operazioni. Esso ha una struttura ciclica in cui si alterna l’esecuzione della operazione speciale con quella esterna, il cui il codice e i dati da elaborare sono stati acquisiti dall’operazione speciale.

**Discutere le motivazioni alla base dei processori multicore**

I microprocessori hanno visto una crescita esponenziale delle prestazioni (organizzazione e frequenza clock). Il miglioramento dell’organizzazione del chip, è stato fortemente focalizzato sul parallelismo: è stata introdotta la pipeline, poi si è passati a pipeline parallele e infine a processori SMT (Simultaneous Multi Threading).

Tutta questa complessità non è però gratis: richiede infatti una logica più complessa, e di conseguenza un’area del chip maggiore per supportare il parallelismo, che però è più difficile da progettare, realizzare e testare.

Inoltre, le CPU SMT erano giunte al limite per quanto riguarda potenza richiesta e calore prodotto, per questo si è scelto di passare alle architetture multicore.

Si è supposto infatti che quest’ultima tipologia di CPU ha il potenziale per ottenere un miglioramento lineare, anche se i vantaggi prestazionali dipendono in parte dai programmi, che devono sfruttare adeguatamente questo parallelismo.

**Si descrivano le possibilità alternative di organizzazione di un processore multicore**

L’organizzazione multicore dipende dal numero di core per chip, dal numero di livelli di cache per chip (L1, L2, L3...) e dalla quantità di cache condivisa. Questo divide i multicore in 4 gruppi: cache L1 dedicata (ogni CPU ha la propria cache L1), cache L2 dedicata (ogni CPU possiede la propria cache L2 (oltre che L1)), cache L2 condivisa (ogni CPU possiede la propria cache L1, inoltre è prevista una cache L2 condivisa tra i core) e cache L3 condivisa (ogni CPU possiede i propri L1 e L2 cache). Inoltre, è presente una cache L3 condivisa tra i core. Le ultime due alternative sono le migliori, in quanto riducono il numero di miss totali. Inoltre, non esiste ridondanza in cache L2 o L3 tra i due core.È anche possibile, tramite appositi algoritmi di sostituzione blocchi, allocare dinamicamente la cache, in modo che ogni core abbia cache dedicata che fornisce un miglioramento alla comunicazione tra processi, anche su core diversi.