DOMANDE ARCHITETTURA – SECONDA PARTE

- ***Si Spieghi in dettaglio la rappresentazione dei numeri reali secondo lo standard IEEE 754.***

IEEE 754 definisce la rappresentazione in virgola mobile più importante.
Questo standard utilizza un formato singolo (32 bit) e un formato doppio (64 bit). Sia il formato singolo che il doppio, prevedono il **primo bit** come **bit di segno**, seguono poi i **bit dell'esponente polarizzato** (8 bit formato singolo e 11 formato doppio) e quelli della **mantissa** (23 bit singolo e 52 bit doppio).

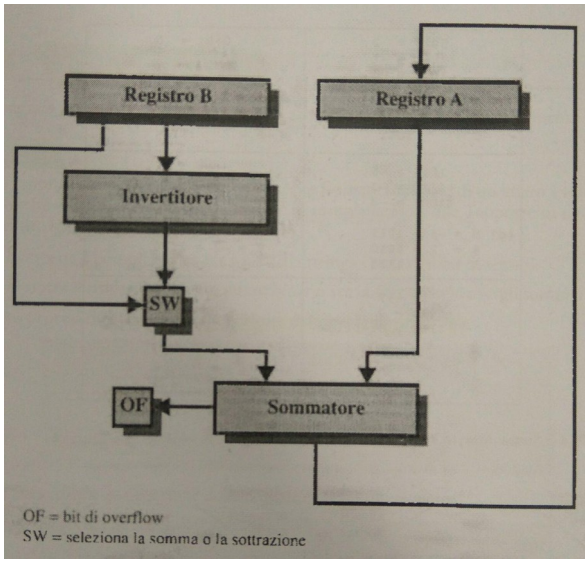
Possiamo inoltre avere altri 2 formati (i cosidetti ***estesi***), che includono bit aggiuntivi per l'esponente e per la mantissa. Grazie a questo , la possibilità di errori dovuti all'eccessivo arrotondamento e all' overflow intermedio. (**pg 345-346)**

- ***Spiegare in dettaglio la tecnica della moltiplicazione tra numeri floating point (IEEE 754)***

Per effettuare la moltiplicazione tra numeri in floating point, è necessario inanzitutto considerare gli **operandi**: se sono 0 , il risultato è 0.
Il passo successivo consiste nel **sommare gli esponenti**, essendo in forma polarizzata è necessario prima **sottrarre il valore della polarizzazione dalla somma** (altrimenti la somma raddopierebbe la polarizzazione!)
Se non sono presenti overflow o underflow dell'esponente, si deve poi **moltiplicare i significandi** (o mantisse), tenendo conto dei loro **segni.** La moltiplicazione viene eseguita come per gli interi.
Successivamente, il risultato viene normalizzato e arrotondato(la normalizzazione potrebbe causare underflow dell'esponente!)
(pg. 350-351)

- ***Si spieghi in dettaglio l' hardware utilizzato per realizzare somma & sottrazione di 2 interi in complemento a 2.***

L'elemento centrale è il **sommatore binario**, al quale vengono forniti gli addendi, dai quali produce una somma e un segnale di overflow.
Il sommatore tratta i due numeri come interi senza segno.
Gli addendi, provengono dai due registri A e B, nei quali può anche essere salvato il risultato. (oppure in un registro apposito)
L' indicazione di **overflow** viene posta in un **flag**.
Per la **sottrazione**, i sottraendo (proveniente dal registro b), viene modificato e posto in complemento a due dal sommatore.
(PG. 329-330)



Erik Nucibella

- ***Si spieghi in dettaglio la codifica in complemento a due degi interi.***
  Si discutano poi i problemi legati alla moltiplicazione di due interi in complemento a 2, esemplificandoli su un caso concreto di moltiplicazione.

Un numero in complemento a 2, è rappresentato:

- Bit più a sinistra indica il **segno**(1=- 0=+);
- Avendo n bit, è possibile rappresentare numeri da -2^{n-1} a $+2^{n-1}$ -1;
- Se il numero da rappresentare è **positivo**? Come in modulo e segno;
- Se il numero è **negativo**, esso viene rappresentato tramite *somma pesata dei bit,* ovvero viene sommato il valore in binario del bit di segno (posto a 1), agli altri bit.
  Questo significa che il numero negativo più grande rappresentabile sarà 1(bit di segno) seguito da n-1 zeri. (il più piccolo sarà 1 segutio da n-1 uni..)

Per rappresentare un numero negativo -k esistono due metodi:

- Viene calcolato k in binario, viene **complementato** e al complemento viene sommato 1;
- Viene calcolato k in binario, vegono scritti gli stessi bit da destra a sinistra sino al primo 1. I numeri successivi vengono complementati;

I problemi legati alla moltiplicazione in complemento a due, nascono dal fatto che uno dei due numeri (o entrambi) può essere negativo.(Se il bit di segno viene calcolato nel prodotto, il risultato sarà errato)

Ex.

5 (1011) x 3 (0011)

Utilizzando i prodotti parziali, otteniamo:

1011 x
<u>0111</u>
1011

<u>1011</u>
100001 che convertito in decimale è **-**31

Per risolvere questo problema, occorre utilizzare la rappresentazione in complemento a 2 per prodotti parziali (***Algoritmo di Booth)***

(PG 322-330)

- ***Si spieghi in dettaglio la differenza tra la codifica in complemento a 2 e quella modulo e segno.***

Nella rappresentazione **modulo e segno,** in una parola d n bit, gli n-1 bit a destra contengono il **modulo** del numero, mentre il bit più a sinistra contiene il **segno** dello stesso (0=+ 1=-).
Il più grande svantaggio di questa rappresentazione è la rappresentazione dello 0, che può essere rappresentato sia come +0(0000) che con -0(1000). Questo diventa difficoltoso per i calcolatori, per questo questa codifica è raramente utilizzata.
Risolve questo problema la rappresentazione in **complemento a due,** che associa il segno al primo bit, mentre i successivi rappresentano il valore del numero:

- Avendo n bit, è possibile rappresentare numeri da -2^{n-1} a $+2^{n-1}$ -1;
- Se il numero da rappresentare è **positivo**? Come in modulo e segno;
- Se il numero è **negativo**, esso viene rappresentato tramite *somma pesata dei bit,* ovvero viene sommato il valore in binario del bit di segno (posto a 1), agli altri bit.
  Questo significa che il numero negativo più grande rappresentabile sarà 1(bit di segno) seguito da n-1 zeri. (il più piccolo sarà 1 segutio da n-1 uni..)

Per rappresentare un numero negativo -k esistono due metodi:

- Viene calcolato k in binario, viene **complementato** e al complemento viene sommato 1;
- Viene calcolato k in binario, vegono scritti gli stessi bit da destra a sinistra sino al primo 1. I numeri successivi vengono complementati;

(PG 321-322)

- ***Si illustrino in dettaglio i possibili approci di ritorno da una chiamata di procedura.***

La procedura è forse stata la più grande innovazione nei linguaggi di programmazione. In ciascun punto del programma essa può essere invocata, la CPU la esegue e poi ritorna al punto in cui si è verificata la chiamata.

Il funzionamento di essa, richiede due dettagli base: una **chiamata** che provoca il salto, e una di **ritorno**, che fa tornare al punto dove c'è stata la chiamata.

Esistono 3 modi efficaci per trattare il ritorno da una procedura:

- **Usare un registro:** Se viene adottato questo approcio, CALL X(*chiamata di procedura)* provoca il salvataggio nel **registro RN** DEL **PC** e della **lunghezza dell'istruzione**. La chiamata di ritorno userà i dati salvati in **RN**;
- **Memorizzare l'indirizzo a inizio procedura:** La procedura salva l'indirizzo di ritorno all'interno di essa. Ciò è pratico e sicuro;
- **Usare la cima di una pila:** Questo è decisamente l'approcio più potente e funzionale. Quando la CPU esegue la chiamata, posiziona l'indirizzo di **ritorno** nella cima della pila, e quando esegue il ritorno, utilizza esso.
-

L’ ultimo approcio è decisamente il più funzionale poiché, oltre a trasmettere l’indirizzo di ritorno, la procedura deve anche **salvare dei parametri**, che anziché venire salvati in registri, possono invece posizionti in coda alla pila.
(PG. 386-390)

- ***Si descrivano nel dettaglio le modalità di indirizzamento con spiazzamento e a pila.***
  In particolare, si confrontino criticamente i 2 metodi di indirizzamento e se ne discutano pregi e difetti.

Il metodo di indirizzamento con **spiazzamento**, combina l’indirizzamento diretto con quelle del registro indiretto.
Questa tecnica richiede che l’istruzione abbia due campi indirizzo, e che almeno uno sia esplicito.

EffectiveAddress= Address + (Address Register)

o più semplicemente **EA= A +R**

Abbiamo 3 tipi di indirizzamento con spiazzamento:

- ***Relativo:*** Dove **R** è il **PC** quindi il mio indirizzo effettivo è **EA=A+PC**
- ***Registro-base:*** Il campo indirizzo contiene uno spiazzamento, mentre R contiene il puntatore all’indirizzo.
- ***Indicizzazione:*** Il campo **A** contiene l’indirizzo della memoria centrale, mentre il campo **R** contiene lo spiazzamento positivo da tale base. Si noti che l’ indicizzazione è l’**opposto** della registro-base. Comodo per elenchi di dati.

L’indirizzamento a **pila**(sequenza lineare di locazioni riservate in memoria), utlizza un puntatore che è contenuto nel registro **SP(StackPointer)**,il cui valore è l’indirizzo nella cima della pila. Questa tecnica è di fatto una forma di **indirizzamento a registro indiretto**; le istruzioni macchina operano implicitamente sulla cima della pila, senza accedere alla memoria.

Il **vantaggio** dello **spiazzamento** è senza dubbio la **grande mole di dati** che si possono **indirizzare** anche se gli accesi alla memoria possono rallentare il metodo.
La **pila non presenta accessi alla memoria**, in quanto il riferimento è implicito ma non è possibile applicare tale metodo a tutto il sistema.

(PG. 422-424)

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

Il formato di un’istruzione definisce la disposizione dei suoi bit in termini di parti costituenti, e deve includere (implicitamente o meno) l’ **OPCODE**.
Il problema fondamentale è la **lunghezza** delle istruzioni: essa infatti dovrebbe essere **uguale alla larghezza del bus di memoria o l’una dovrebbe essere multiplo**

dell'altra. Inoltre, dovrebbe essere multiplo della lunghezza di un carattere, che di solito è di 8 bit.

Il **formato** di un istruzione può essere a lunghezza:

1. ***Fissa:*** Ovvero tutte le istruzione hanno la stessa lunghezza,(posso avere più formati cambiando i campi) Estremamente efficiente nel caso di pipeline.
2. ***Variabile:*** Ogni istruzione ha una lunghezza che dipende dal numero degli operandi(nel campo opcode devo specificare il numero di essi), permettono grande flessibilità ma incrementano la complessità;
3. ***Ibrida:*** Ho diversi formati con lunghezza fissa ma diversa.

Le istruzioni a lunghezza variabile o ibrida sono sicuramente più flessibili, permettendo più modi di indirizzamento e più riferimenti, ma incrementano di molto la complessità dell' hardware(CPU). Queste caratteristiche si avvicinano di più alla filosofia **CISC.**

Dall'altro lato, una lunghezza fissa ha come problema la mancanza di operandi indipendenti dall' OPCODE, ne consegue che sono disponibili meno operandi da utilizzare per le operazioni. Vi è in generale mancanza di flessibilità, ma la CPU ne guadagna in termini di complessità e costi, inoltre il caricamento di un' istruzione avviene in modo più veloce rispetto alle istruzioni a lunghezza variabile, ed una dimensione fissa favorisce l'uso della pipeline. Queste caratteristiche si avvicinano alla filosofia **RISC.**

(PG. 430-438)

- ***Si spieghi in dettaglio in cosa consiste il formato variabile delle istruzioni. Dare esempi di formati variabili***

Per formato variabile delle istruzioni, si intende che le istruzioni di una CPU possono avere lunghezze variabili. L'indirizzamento può quindi essere più flessibile e con più operandi . Il prezzo principale da pagare è l'incremento della complessità del processore.

Esempi di formati variabili sono:

1. ***PDP-11:*** E' stato progettato per fornire un linguaggio macchina potente e flessibile per un minicomputer a 16 bit.
   Esso dispone di 8 registri generici da 16 bit.
   Solitamente le istruzioni sono lunghe una parola (16 bit).
   Il linguaggio e la complessità del PDP-11 sono complesse, questo aumenta i costi lato hardware.
2. ***VAX:*** Per progettare questo formato, sono stati rispettati 2 criteri fondamentali:
   - Tutte le istruzioni dovrebbero avere un numero *naturale* di operandi;
   - Tutti gli operandi dovrebbero presentare la stessa generalità nelle specifcihe;

     L' OPCODE può stare su 1-2 byte.

   E' un sistema molto flessibile e potente che facilita il lavoro di programmatori e compilatori, ma il sistema è molto complesso.

(PG. 435-440)

Erik Nucibella

- ***Spiegare in dettaglio i fattori che influenzano la lunghezza delle istruzioni di una CPU.***

La lunghezza delle istruzioni condiziona (ed è condizionata) da:

- **Dimensione della memoria;**
- **Organizzazione della memoria;**
- **Struttura del bus;**
- **Complessità della CPU;**
- **Velocità della CPU;**

  Il compromesso che si cerca è ovviamente quello di avere un repertorio di istruzioni vasto e potente, e la necessità di risparmiare spazio.

Più codici operativi e operandi, semplificano infatti la vita del programmatore ,ma ovviamente aumentano le dimensioni della memoria.

La lunghezza delle istruzioni dovrebbe essere **uguale** (**o multipla**) alla dimensione del **bus di memoria**. Inoltre, essa dovrebbe essere anche **multipla** della lunghezza di un *char ,* quindi **di 8 bit**.

(PG. 430-431)

- ***Contesto Pipeline: Che cos'è e come funziona lo sbilanciamento delle fasi?***

Lo Sbilanciamento delle fasi è uno dei **problemi** legati alla pipeline, dovuti alla diversa durata delle fasi della stessa. In particolare, questo accade poiché non tutte le istruzioni richiedono le **stesse fasi e risorse**, e non tutte le fasi richiedono lo stesso **tempo di esecuzione**.
Per evitare questo problema, è possibile:

- Introdurre **tempi di attesa** forzati;
- Decomporre fasi onerose in **più sottofasi**;
- **Duplicare gli esecutori** delle fasi più onerose e farli lavorare in parallelo.

- ***Contesto Pipeline: Discutere il problema della dipendenza dei dati, e le tecniche per trattare il problema.***

Un Hazard dei dati (o dipendenza dei dati) si verifica quando esiste un conflitto nell'accesso alla locazione di un operando, in particolare, **entrambe accedono** a uno specifico indirizzo di memoria o registro.
Se questo accadesse in sequenza non ci sarebbero problemi, ma dato che le istruzioni sono eseguite in pipeline, questo potrebbe portare a **risultati diversi** da quelli aspettati.

Erik Nucibella

I tipi di hazard dati sono:

- **RAW(ReadAfterWrite):** Quando un'istruzione modifica un registro o una locazione di memoria, e un'istruzione successiva legge il dato in quella locazione. Si verifica un hazard se la lettura avviene prima del completamento della scrittura.
- **WAR(WriteAfterRead):** Si verifica un hazard se, avendo un'istruzione che legge una locazione di memoria e un'istruzione successiva che vi scrive, la scrittura avviene prima della lettura.
- **WAW(WriteAfterWrite):** Due istruzione devono scrivere nella stessa locazione. Vi è un hazard se le scritture avvengono in modo invertito.

Per gestire questo problema, esistono diverse soluzioni:

- Introduzione di **NOP(NotOperativePhase),** O **stallo**;
- Prelievo del dato direttamente dall'uscita della ALU (**Data Forwarding**);
- Risoluzione a livello di **compilatore**;
- Riordino delle istruzioni o **Pipeline Scheduling**;

(PG. 473 IN POI)

- ***Contesto Pipeline: Spiegare in dettaglio la dipendenza dal controllo, e in particolare la tecnica del buffer circolare.***

Un Hazard di controllo (o Hazard di salto) si verifica quando la normale esecuzione della pipeline viene alterata (per esempio da un salto condizionato).
Quando il **PC viene alterato**, la pipeline viene invalidata.

Per gestire questo si può:

- Mettere in **stallo** la pipeline finché non si ha l'indirizzo della prossima locazione;
- Individuare le **istruzioni critiche** per anticiparne l'esecuzione, con tecniche come *flussi multipli, prelievo anticipato della destinazione e buffer circolare.*

Il **buffer circolare** in particolare, è una memoria piccola e molto veloce, gestita nella fase di fetch delle istruzioni, che **contiene le ultime n istruzioni** da prelevare.
In caso di salto, si controlla se la destinazione è già presente nel buffer, **evitando** così il **fetch**.
Il riconoscimento delle istruzioni avviene in modo molto **simile** a quello che accade in **cache**.
I vantaggi di questa tecnica sono:

1. Si può anticipare il fetch di alcune istruzioni successive a quella corrente portandole nel buffer: Se non vi è salto, non c'è nessun problema.
   Se invece si salta in avanti di poche istruzioni, l'istruzione sarà quasi sicuramente già nel buffer.
2. Se sono presenti cicli, iterazioni o simili, il buffer dovrà prelevare solo una volta le istruzioni , per quelle successive, esse sono già nel buffer.

(PG. 475-76)

Erik Nucibella

- ***Contesto Pipeline: Spiegare in dettaglio tecnica di predizione del salto utilizzando 2 bit di controllo.***

La tecnica di predizione del salto, cerca di verificare se un salto sarà preso o meno, essa funziona associando uno o più bit che codificano la *storia* recente(memorizzati in una locazione ad accesso molto veloce!)
Gli **approci dinamici di predizione**, cercano di migliorare la predizione memorizzando la *storia* delle istruzioni di salto di uno specifico programma.

In particolare, la predizione dinamica a **2 bit** funziona nel seguente modo:
Avendo 2 bit a disposizione, è possibile utilizzare 4 diversi stadi per trattare i salti:

- **00:** Si ipotizza che avverrà un salto, in caso di salto ci si sposta sullo stato **01**, altrimenti si resta in **00**(errore di predizione)**;**
- **01:** Si ipotizza avverrà un salto, in caso positivo ci si sposta in **00**,altrimenti in **10**(errore di predizione);
- **10:** Si ipotizza no avverrà un salto, in caso di salto ci si sposta in **11**(errore di predizione), altrimenti si resta in **10**;
- **11:** Si ipotizza non avverrà un salto, in caso di salto ci si sposta in **00**(errore di predizione), altrimenti si passa allo stato **10**;

Sostanzialmente, la situazione iniziale è quella nello stato **00;** se successivamente avvengono **2 errori** di predizione in **successione**, ci si sposta nello stato che **prevede l' opposto** di quello previsto finora.

(PG. 477-78)

- ***Contesto Pipeline: Si spieghi la tecnica che utilizza la tabella della storia dei salti. A cosa serve? Discuterne dettagliatamente***

La tabella di storia dei salti è una piccola memoria cache associata allo stadio di IF. Ciascuna riga della tabella è costituita da 3 elementi: **indirizzo di istruzione del salto**,un certo numero di **bit di storia** (conservano lo stato di tale istruzione) e informazioni inerenti all'**istruzione destinazione**(o l'indirizzo istruzione o l'istruzione stessa).

(PG 477-480)

- ***Contesto Pipeline: Si spieghi in dettaglio la tecnica del salto ritardato, fornendo un esempio di applicazione.***

Il salto ritardato è un modo per incrementare l'efficienza della pipeline, esso fa uso di un salto che non ha effetto fino al termine dell'istruzione seguente.(da qui **ritardato**) La CPU, infatti esegue **sempre** l'istruzione di salto, e solo in segutio altera (se necessario) la sequenza di esecuzione delle istruzioni. La locazione successiva a quella di salto viene detta ***delay slot.***

Pg(526-527)

- ***Contesto Pipeline: Si spieghi in dettaglio la tecnica del data-forwarding. A cosa serve? Di che hardware ha bisogno? Come funziona?***

Il data-forwarding è una tecnica che cerca di ridurre il problema di **hazard dei dati**.Individuata la dipendenza, esso consiste nel prelevare il dato richiesto direttamente all'**uscita** della **ALU**, attraverso appositi **ciruti di bypass e MUX**(regolati da unità di controllo e altre unità).
Nel caso di architettura **MIPS**, sono presenti circuiti **EX --→ EX** e **MEM --→ EX**.
Questa soluzione riduce notevolmente il numero di stalli di un'istruzione, e di conseguenza anche i cicli di clock che la pipeline impiega.
Per mandare i dati dall'ALU alle istruzioni, si potrebbe pensare di utilizzare un nuovo **multiplexer** da mettere davanti ad ogni ingresso in ALU. Questo consentirebbe di verificare se occorre seguire il percorso normale di istruzioni, o di inoltrare il dato nel circuito di bypass. Nella realtà, il circuito di data-forwarding può avere **diverse implementazioni**, ma solitamente abbiamo 3 componenti fondamentali:

- **Forwarding unit:** Decide se attivare il bypass attivando opportunamente MUX e multiplexer;
- **Hazard Detection Unit:** Riconosce le dipendenze, genera stalli se esse non sono risolvibili;
- **Control Unit:** Manda segnali di controllo che regolano il forward dei dati.

(Fonte: Pipeling MIPS in English)

- ***Spiegare in che modo un compilatore possa aiutare l'utilizzo efficace dei registri da parte di un architettura RISC.***

L' approcio software per ottimizzare l'**uso dei registri**, consiste nell'utilizzo del compilatore. Esso, per massimizzare l'utilizzo dei registri cercherà di allocare nei registri le **variabili maggiormente utilizzate** in un dato intervallo temporale. Questo, richiede l'utilizzo di sofisticti algoritmi per **analizzare i programmi**.

Per effettuare questo, viene effettuata una mappatura, che equivale a risolvere il probelma di ***colorazione di un grafo***:
Il compilatore assegna un registro simbolico alle variabili candidate, quindi mappa un numero (virtualmente) illimitato di registri, utilizzando registri e reali , ed eventualmente lo spazio in memoria centrale.

Quindi, l'essenza dell'otimizzazione tramite compilatore, è quella di risolvere questo grafo, secondo queste regole:

Dato un grafo, costituito da **registri simbolici** connessi tra loro
- Si assegni un colore per ogni registro, in modo che:
  - Nodi adiacenti abbiano lo stesso colore;
  - Si utilizzi il minor numero di colori possibili;
- Due registri all'interno dello stesso codice sono **collegati** da archi;
- Idea di fondo: **colorare** il grafo con n colori, dove **n** è il numero di **registri reali**
- Nodi che non possono essere colorati vanno messi in memoria centrale;

(PG. 512-518)

Erik Nucibella

- ***Si motivi la presenza nei processori RISC, di un ampio banco di registri ad uso generale. Si spieghi in dettaglio funzionamento e meccansimo di tali registri.***

La ragione per cui le CPU RISC utilizzano un ampio banco di registri ad uso generale, è essenzialmente perché essi hanno bisogno di mantenere al loro interno il maggior numero di variabili per lunghi periodi, **minimizzando** così l'**accesso in memoria**. L' utilizzo dei **registri rispetto** ad altre memorie, è dato dal fatto che essi sono i più veloci dispositivi di memorizzazione presenti(superiori a cache).
Fisicamente, sono previsti 3 tipi di registri adibiti a questo compito:

- **Registri dei parametri:** Conengono i paramentri passati dalla procedura chiamante a quella corrente, mantengono i risultati da restituire;
- **Registri locali:** Vengono utilizzati dalle variabili locali, come deciso da compilatore;
- **Registri temporanei:** Sono usati per scambiare parametri e risultati con il livello inferiore(la procedura chiamata)

(PG 512-513)

- ***Si metta confronto il modo in cui la RISC usa l'ampio banco di registri a disposizione, rispetto alla gestione di una cache.***

Il banco dei registri organizzato in finestre, agisce come un piccolo e rapido buffer che conserva le variabili che hanno maggiore probabilità di essere usate con più frequenza. Da questo punto di vista il banco di registri **somiglia a una memoria cache.**
Ci sono però alcune sostanzali differenze da considerare:

- **I registri** conservano **tutti gli scalari locali**. La **cache** conserva solo quelli **usati di recente**.
- **I registri** contengono **solo le variabili in uso.** La **cache** opera invece usando **blocchi**, che potrebbero non essere utilizzati totalmente.
- **I registri** contengono le **variabili globali indicate dal compilatore**, ma è difficile per il comilatore determinare quali variaibli verranno usate in modo intensivo; La **cache** invece, mantiene solo quelle **usate di recente**;
- Il **trasferimento dati** tra **registri** e **memorie** è poco frequente, mentre quello della **cache**, dipende dall'algoritmo usato dalla stessa;
- **I registri** utilizzano un **indirizzamento registro**, molto più veloce rispetto all' **indirizzamento memoria** della cache.

(PG. 515-516)

Erik Nucibella

- ***Si spieghino in dettaglio le motivazoni alla base dell'architettura RISC.***

L' architettura RISC nasce in base all' esigenza di avere un **costo hardware minore**, e processori quindi più semplici, e solitamente reattivi.

Le principali caratteristiche di quest'architettura sono:

- **Un'istruzione per ciclo**, ovvero un'istruzione macchina per ogni **ciclo macchina**(tempo per prelevare 2 operandi + eseguire operazione LAU + memorizzare risultati);
- **Operazioni registro-registro:** Le operazioni coinvolgono quasi sempre operandi presenti nei registri, quindi ad accesso **molto veloce.**
- **Semplcici modi di indirizzamento:** Quasi tutte le istruzioni RISC utilizzano un **indirizzamento a registro**, altri modi possono essere derivati via software. Questo semplifica l'insieme istruzioni e l'unità di controllo.
- **Semplici formati delle istruzioni:** E' presente un unico formato(o al massimo due); La **lunghezza istruzioni** è fissata, come le **posizioni dei campi.** Questo porta a diversi vantaggi:
  - la **decodifica OPCODE e l'accesso operandi** possono avvenire **simultaneamente;**
  - Più semplice sviluppare compilatori efficienti;

Altri vantaggi di questa architettura, sono per esempio il fatto che la maggior parte delle **istruzioni** generate dal compilatore sono **semplici**, oppure il fatto che le CPU RISC sono maggiormente reattivi agli **interrupt**.

(PG. 520-522)

- ***Si spieghino in dettaglio le motivazoni alla base dell'architettura CISC.***

L' architettura CISC, nacque per diversi motivi, anche se alcuni dei quali successivamente si sono rivelati errati.

Essi sono:

- **Semplificazione** dei **compilatori**;
- Programmi (teoricamente) più **piccoli e veloci**, infatti sicuramente il programma sarà **più breve**(composto da meno istruzioni), ma non è sicuro che sià più piccolo in termini di dimensioni;
- Linguaggi macchina più complessi dovevano produrre **codice eseguibile** più **rapidamente**(in realtà non è così);
- **Supportare i linguaggi ad alto livello** (sempre più complessi);
- **Facilitare** il lavoro del **programmatore** a **discapito** del costo dell'**hardware**;
- La memoria di controllo agisce come cache per le istruzioni;

I punti su cui puntava l'architettura CISC, si sono presto rivelati non del tutto corretti e fondati, infatti la tendenza verso sistemi più complessi, non sempre porta a prestazioni migliori.

(PG. 519-520)

- ***Si spieghino in dettaglio le differenze tra CISC e RISC.***

Il confronto CISC – RISC, contrariamente a quello che si può pensare, non vede un vincitore netto.
Al giorno d'oggi infatti, si è visto che i processori RISC possono trarre vantaggio da alcune caratteristiche RISC e viceversa.
Per esempio, il PowerPC non è un RISC *puro.* Dall' altro lato, il Pentium di Intel incorpora caratteristiche RISC.

La filosofia **RISC** comunque, predilige le **istruzioni primitive**, mentre la **CISC** utilizza **istruzioni complesse**.
A cause di questo, la **CISC** deve avere un'**unita di controllo più complessa** e una **memoria del microprogramma di controllo più ampia**.

Dall' altro lato, il **RISC** utilizza **pochi metodi di indirizzamento**(prediligendo quello a registro), Usa un **formato** istruzioni **fisso** e una **dimensione** generalmente multipla di **4 byte**.

Questo indica spesso una **difficoltà** di **decodifica** delle **istruzioni**, e una **facilità** del **pipeling** di istruzioni.

(PG. 518-524)

- ***Si spieghi come l'architettura RISC può trattare efficientemente la chiamata annidata di procedure.***

Tipicamente, le chiamate di procedura coinvolgono **pochi parametri**, e presentano una **basso grado di annidamento.** Questo permette di usare molti gruppi di registri, detti **finestre di registri** per risolvere il problema:

- Una chiamata di procedura seleziona automaticamente una nuova finestra;
- Il ritorno da una procedura, seleziona la stessa finestra assegnata precedentemente alla procedura.

La finestra, è divisa in **3 sottogruppi**:

- **Registri dei parametri:** Contengono i parametri passati dalla procedura chiamante a quella corrente;
- **Registri locali:** Che memorizzano il contenuto delle variabili locali di procedura;
- **Registri temporanei:** Usati per scambiare risultati e parametri con il livello inferiore, gestiscono il ritorno da una procedura.

Registri temporanei di una finestra, si sovrappongono con quelli che contengono i parametri successivi, cioè quelli della finestra riferita a una chiamata annidata. Questi registri sono fisicamente gli stessi, ciò consente il **passaggio** di **variabili senza**

accedere alla memoria. L'organizzazione fisica del banco di registri è un **buffer circolare** di finestre che si sovrappongono. Quando avviene una chiamata di procedura, il puntatore **CWP** viene aggiornato per farlo puntare alla finestra di procedura. In caso di molte procedure annidate, se il **buffer** è **pieno**, viene salvata in memoria la prima finestra inserita, e **sovrascritta** da quella corrente. Quando la procedura termina, grazie al puntatore **SWP** è possibile **ripristinare** la finestra che è stata salvata in memoria principale.

(PG. 512-513)

- ***Contesto Pipeline MIPS: Spiegare come lo stadio ID è in grado di identificare dipendenza dei dati.***

Quando un'istruzione passa dalla fase ID a quella EX, si dice **rilasciata**(**issued**). Nella pipeline MIPS, è possibile individuare tutte le dipendenze dei dati nello stadio ID. Se si rileva un hazard dei dati per una istruzione, questa va in **stallo** ancor prima di essere rilasciata. Sempre in questa fase, è possibile determinare che tipo di **data-forwarding** adottare per evitare lo stallo. La logica per decidere come effettuare il forwarding dei dati, è simile a quella appena vista, ma considera molti più casi. Osservazione importante è che o registri pipeline contengono:

- Dati su cui effettuare il forwarding;
- Campi registro sorgente & destinazione;

I dati su cui effettuare il forwarding provengono o dall' **output ALU** o dalla memoria principale; essi sono diretti verso l'**input ALU** o l'input della memoria dati.

(Slide MIPS)

- ***Descrivere i 3 tipi di istruzione MIPS.***

L'architettura MIPS che è u tipo di RISC con pipeline ottimizzata, prevede 3 tipi di istruzioni o *formato istruzioni:*

- Formato **R(Registro):** Sono 32 registri da 32 bit; le operazioni avvengono sempre tra registri, quindi questo registro è utilizzato per le **operazioni logio-aritmetiche**;
- Formato **I(Istruzioni load/store):** Sono istruzioni per trasferire dati tra memoria e registri, può essere anche utilizzato per operazioni logico-aritmetiche se l'**operando** è **immediato**.
- Formato **J(Jump):** Per le istruzioni di **salto**;

Il formato fisso delle istruzioni facilita le operazioni di fetch e decode.

(Slide MIPS)

- ***Si descriva sinteticamente l'implementazione delle istruzioni attraverso la tecnica della microprogrammazione. (Processori RISC o CISC?)***

L'implementazione delle istruzioni attraverso la tecnica della microprogrammazione, si ottiene facendo corrispondere l'**OPCODE** all'**indirizzo di inizio** di un **microprogramma**. Ad ogni istruzione macchina, viene associato un microprogramma formato da una **sequenza di microistruzioni**. Esse sono formate da **microordini**(ognuno corrispondente ad un segnale di controllo) registrati nella **ROM**(detta anche **memoria di controllo)**, in una word detta **word di controllo**. In questa word, ogni bit corrisponde ad un microordine, ovvero rappresenta una linea di controllo. Le unità microprogrammate si possono suddividere in due categorie:

- **microprogrammazione orizzontale:** microistruzioni con **numero elevato** di **bit**, possono svolgere **svariati compiti** in **parallelo**, generando svariati segnali di controllo;
- **microprogrammazione verticale:** microistruzioni presentano un **numero limitato** di **bit**, questo conduce ad una **minore velocità** di **funzionamento**, (microistruzioni per specificare quanto fatto con una orizzontale).

La microprogrammazione è tipica di architetture **CISC** per implementare l'unità di controllo, in quanto essa permette una **maggiore flessibilità** in fase di **progettazione**, rendendo più **facile modificare** le sequenze di **microoperazioni**.

- ***Si spieghino in dettaglio le motivazioni alla base dei processori multicore.***

I microprocessori, hanno visto una crescita esponenziale delle prestazioni (**organizzazione & frequenza clock**); il miglioramento dell'organizzazione del chip, è stato fortemente focalizzato sul **parallelismo:** E' stata introdotta la pipeline, poi si è passati a pipeline parallele e infine a processori SMT (Multithread simultaneo). Tutta questa **complessità** non è però gratis: essa infatti richiede 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, abbia il potenziale per ottenere un miglioramento lineare, anche se i **vantaggi** prestazionali **dipendono** in parte dai **programmi**, che devono **sfruttare** adeguatamente questo **parallelismo**.

Erik Nucibella

- ***Si illustrino le possibili alternative di organizzazione di un processore multicore.***

L’organizzazione multicore dipende da:

- **Numero di core per chip;**
- **Numero di livelli di cache per chip** (L1,L2,L3...)**;**
- **Quantità di cache condivisa;**

Questo divide i multicore in 4 gruppi:

- **Cahce 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**;**
- **Cache L3 condivisa**: Ogni CPU possiede i propri L1 e L2 cache. Inoltre, è presente una cache L3 condivisa tra i core**;**

Ovviamente, le ultime due alternative sono le migliori, in quanto esse **riducono** il numero di **miss totali**, inoltre **non esiste ridondanza** in cache L2 o L3 tra i due core. E’ anche possibile, tramite appositi algoritmi di sostituzione blocchi, allocare **dinamicamente** la **cache**, in modo che ogni core abbia cache **dedicata**.
Cache dedicata fornisce inoltre un miglioramento alla **comunicazione tra processi**, anche su core diversi.

Erik Nucibella