Appunti delle Lezioni di Calcolatori Elettronici – Ing. Informatica tenute dal Prof. Giuseppe Ascia

Le lezioni sono state trascritte ad opera degli studenti

Alessio Oglialoro.
Luis Antonio Previti.
Salvatore Consolato Meli.
Giuseppe De Luca.
Giuseppe Di Prima.
Gregory Callea.
Maria Grazia Di Sì.

Lezione 18 aprile 2011
Parleremo di componenti che sono presenti all'interno di un sistema digitale, in particolare all'interno di un processore. Questi componenti sono utili per costruire un sistema di elaborazione, nel caso specifico la path del sistema di elaborazione che è il processore. Quest'ultimo è caratterizzato da due componenti : (cfr.appunti block notes per capire cosa scrive alla lavagna)

- l'unità di controllo che genera dei segnali per il data path, che sostanzialmente è la parte di un processore che contiene tutto ciò che serve per realizzare le operazioni. Sulla base di certi ingressi, quali per esempio la memoria, l'U.C. dice al data path cosa bisogna fare e quest'ultimo produce certi risultati sulla base anche di certe condizioni che vengono inviate dal data path e che sostanzialmente sono i risultati d'elaborazione che vengono monitorati dall'unità di controllo e sulla base di questi risultati l'unità di controllo decide cosa fare.

In questo schema a macro blocchi vediamo un sistema costituito da due parti, che a loro volta conterranno tutta una serie di componenti che servono per realizzare sia l'unità di controllo che il data path. Gran parte dell'unità di controllo è dovuta ad una macchina a stati, quindi la rete sequenziale che realizza una macchina a stati e quindi qui dentro saranno necessari dei registri e delle reti combinatorie che servono per realizzare la rete sequenziale. Per quanto riguarda il data path, avremmo dei registri e una serie di componenti elementari quali multiplexer, decodificatori, l'ALU e altri componenti che messi insieme permettono di realizzare ed eseguire le istruzioni.

Componenti che ci permettono di realizzare il data path, ovvero la parte di elaborazione che è quindi in grado di realizzare le operazioni vere e proprie all'interno del sistema digitale. Il primo componente preso in considerazione e peraltro estremamente semplice è il multiplexer o selettore. Consideriamo un caso semplicissimo: due ingressi i1 e i0, un uscita u e un segnale s. Questo multiplexer sulla base del segnale di controllo, di selezione, farà in modo che l'uscita assuma o il valore d'ingresso i1 o i0, quindi di fatto l'uscita u è uguale a uno dei due valori in ingresso. Volendo dare una descrizione in pseudo C il comportamento sarà il seguente:

se s=0 allora u=i0 oppure u=i1. (cfr. appunti per disegno e formule) Questa è la descrizione comportamentale di questo componente. Se vogliamo dare una descrizione tramite tabella della verità dovremmo in questo caso indicare per ognuno di questi otto valori d'ingresso qual'è il valore della corrispondente uscita. Abbiamo le otto configurazioni e quindi in questo caso l'uscita coincide con i1 o i0 sulla base del valore s : quando è 0 coincide con i0, quando è 1 coincide con i1, ecc.. (cfr.tabella sugli appunti) Da questa tabella della verità applicando metodi che già conosciamo siamo in grado di determinare il valore dell'uscita. (va inserita qui un'altra tabella) l'uscita u sarà pari in questo caso a s negato i0 + s i1. Questa è l'espressione. Sostanzialmente quando s è zero attivo i0, quando s è uno attivo i1. Quanto detto vale nel caso di multiplexer 2x1(due ingressi e

un'uscita a singolo bit). A partire da questo noi possiamo costruire anche multiplexer 2x1 che anzicchè essere a 2 singoli bit sono formati da due vettori i1 e i0 con uscita s: questo multiplexer 2x1 come viene ottenuto considerando un vettore a n bit? Prendiamo come caso semplice quello con un vettore di due bit, basterà allora mettere in parallelo due multiplexer : in uno mandiamo ad esempio i11 e i01 quindi il bit uno del vettore, nell'altro mandiamo i10 e i00, le uscite saranno u1 e u0 e a entrambi viene mandato lo stesso segnale di selezione s, quindi in generale se noi abbiamo due vettori d'ingressi di dimensioni n basterà costruire n istanze dello stesso mux per costruire un mux 2x1 con ingressi di n bit; la realizzazione è estremamente semplice perchè questo è uno dei componenti più semplici che si possono utilizzare per costruire il data path. Ci sono però situazioni in cui anzicchè avere due soli ingressi si hanno più ingressi, per esempio, potremmo avere 4 ingressi e un'uscita : in questo caso il mux (vedi disegno appunti) 4x1 avrà bisogno per individuare qual'è l'ingresso da mandare in uscita di due bit di selezione, perchè dati 4 ingressi per selezionarne uno serve un log in base 2 del numero di ingressi come numero di bit per la selezione, con 2 ingressi bastava un solo bit, per 4 ingressi ne servono 2, per 8 ne servono 3. Per quanto riguarda la descrizione questa potremmo scriverla in pseudo C: se s1=0 and s0=0 allora uscita=i0 altrimenti se s1=0and s0=1 l'uscita è pari a i1, altrimenti se s1=1 and s0=0 l'uscita è i2 infine ultimo caso ovvero entrambi sono a uno allora l'uscita è i3. Stavolta, senza far uso di tabelle di verità perchè sarebbe pesante da scrivere in quanto bisognerebbe fare i cubi con 6 variabili, prendendo in considerazione l'esempio del mux 2x1 dovremmo essere in grado di scrivere direttamente l'espressione relativa all'uscita : nel caso di mux 2x1 avevamo scritto

if s=0 segue u=i0 else u=i1; l'espressione era stata invece u=s' i0 + s i1. Ora guardando le condizioni del mux 4x1, dall'espressione scritta per il 2x1 e dallo pseudo C scritto per il mux 4x1, otteniamo l'espressione per il mux 4x1 che sarà : u= s1' s0' i0+s1' s0 i1+...quindi basta considerare la coppia s1 s0 in forma diretta o negata, la configurazione di questa coppia di segnali di controllo corrisponde al pedice di i in binario : se abbiamo 0,0 allora 0, se 0,1 allora 1. In questo modo si tira fuori l'espressione del mux come somma di prodotti. Il mux 4x1 è ottenuto considerando due soli livelli in quanto abbiamo una somma di prodotti. Supponendo di voler realizzare questo mux 4x1 senza andare a fare la sintesi di questa rete combinatoria, ma lo vogliamo realizzare utilizzando un mux 2x1 che già è disponibile. Normalmente nella progettazione dei sistemi digitali là dov'è possibile per rendere molto più veloce la progettazione di un sistema si utilizzano delle librerie di componenti che già sono state progettate. Se abbiamo a disposizione un componente mux 2x1 già pronto e progettato e noi dobbiamo solo utilizzarlo; utilizzando il mux 2x1 possiamo essere in grado di costruire un mux 4x1 (cfr. disegno appunti) : mandiamo in parallelo le linee su due mux 2x1 (i3 e i2 sul mux 1 e gli ingressi i1 e i0 al mux 0). A questo punto con questa configurazione dovremmo mandare un segnale di controllo su ciascuno dei due mux. La configurazione i3 è 1,1 e 1,0; i2 è invece 0,1 e 0,0. Se pensiamo a questo mux 4x1 in pratica il segnale si controllo s1 mi definisce se devo considerare o questa coppia o quest'altra perchè se s1=0 allora nel mux 4x1 dovremmo scegliere o i0 o i1, se invece s1=1 andremo a scegliere o i2 o i3; quindi mediante s1 scegliamo quale delle due coppie prendere, mediante s0 invece scegliamo all'interno della coppia o i2 o i3 per una coppia e o i1 o i0 per l'altra coppia. Il primo segnale di controllo dovrà essere quello che mi definisce quale di questi due elementi prendere ovvero quello corrispondente a s0, quindi s0 lo manda in entrambi e a questo punto l'uscita u1 seleziona o i2 o i3 oppure per l'altro mux o i1 o i0. Questi due segnali (cfr appunti) li vado a mandare ad un altro mux 2x1 e in questo caso il segnale di controllo sarà s1 che mi dice in pratica se devo prendere un elemneto di una o l'altra coppia. In questo modo utilizzando il metodo di klussegery riesco senza fare la sintesi di una rete combinatoria a costruire un mux 4x1. Il vantaggio principale di questa soluzione è quello di sfruttare dei

componenti già esistenti riuscendo così a costruire più velocemente il circuito finale, lo svantaggio invece consiste in questo : utilizzando una struttura di questo tipo quello che può accadere è che la latenza che si ha con questo tipo di circuito è maggiore rispetto a quella che avremmo andando a realizzare il circuito facendo la sintesi come fatto prima, perchè nel circuito ottenuto nella prima versione avevamo un'espressione come somma di prodotti, avevamo due soli livelli in cui termini erano implicanti con tre letterali, qui invece abbiamo una cascata di due mux e quindi di fatto avremmo 4 livelli, due per ogni mux. Il vantaggio è che in termini di velocità di progettazione si va più veloci con questo 2° tipo di soluzione e dal punto di vista del costo complessivo questa soluzione potrebbe anche in alcuni casi portare ad un costo inferiore. In questo caso abbiamo come operazioni, ad esempio, due prodotti con due letterali e una somma (mux 2x1) quindi in totale 6 prodotti per 2 elementi e tre somme da due elementi, nella soluzione originale avevamo invece 4 termini quindi 4 prodotti ciascuno da tre e poi una somma da 4. E' da vedere quale delle due soluzioni è più conveniente nel momento in cui si vanno a realizzare i componenti a livello di transistor. Bisognerebbe, utilizzando dei tools di sintesi, stimare esattamente quale delle due soluzioni è quella migliore; sicuramente dal punto di vista del ritardo la soluzione con due livelli porterebbe un ritardo maggiore però il grosso vantaggio è che tiriamo fuori il risultato più rapidamente e alcune volte la cosa la più importante nn è quella di tirare fuori la soluzione migliore in assoluto, se per raggiungere la soluzione ottima dobbiamo spendere troppo tempo e quindi arriviamo troppo tardi rispetto ad un concorrente che magari non ha una soluzione buona come la nostra ma leggermente peggiore però è arrivato sul mercato molto prima di noi quel concorrente conquista il mercato e magari diventa il leader di quel settore. Applicare il metodo in cui si utilizzano componenti già belli e pronti è il metodo migliore per velocizzare la progettazione di un sistema perchè a questo punto molto rapidamente si riesce a tirare fuori una soluzione soprattutto quando la complessità del sistema da progettare diventa molto elevata.

Un altro componente che molto spesso viene utilizzato è il decodificatore. Per capire il funzionamento di quest'altro componente consideriamo inizialmente il caso semplice: (cfr. disegno appunti) esso è caratterizzato da un segnale s in ingresso, da due uscite. come funziona: supponendo che si attivi il segnale di enable fa in modo che questo segnale s definisca tra tutte le uscite qul'è l'unica che assume il valore 0 oppure l'unica che assume il valore 1. Abbiamo due possibili versioni : un decodificatore in logica diretta o in logica negata. In questo caso, per esempio, volendolo descrivere in modo comportamentale possiamo dire che se s=0 allora u0=1 altrimenti u0=0, in parallelo per quanto riguarda l'altra uscita abbiamo che se s=0 allora u1=0 else u1=1. In pratica questo segnale s, ignorando momentaneamente il segnale di enable, quando s=0 l'uscita u0=1 e u1=0 quando s=1, u0=0 e u1=1. Quindi sostanzialmente solo uno dei due ingressi vale uno, l'altro varrà zero. Vediamo cosa succede estendendo al caso di 4 uscite: nel caso di 4,8,...n uscite accadrà sempre che solo una delle uscite vale uno e tutte le altre varranno zero. Quanto detto è valido nel caso in cui l'enable non è presente o è come se fosse sempre uno. Capita alcune volte che è presente anche un segnale di enable: in questo caso vogliamo capire come funziona il circuito: se è presente l'enable e vale 1 allora il comportamenteo del circuito è quello appena visto, se invece l'enable è zero allora entrambe le uscite sono a zero, quindi nessuna delle due uscite è abilitata indipendentemente dal valore del segnale s. Una descrizione del comportamento del circuito in caso di enable=0 in pseudo C potrebbe essere la seguente: if enable =1 & s=0 allora u0=1 else u0=0; if enable=1 & s=1 allora u0=0; enable=0 & s=0 allora u0=0; enable=0 & s=1 allora u0=0. In definitiva solo nel primo caso, cioè enable=1 e s=0, u0=1 in tutti gli altri casi l'uscita vale 0. Altro caso: se enable=1 e s=1 allora u1=1 else u1=0. Dal punto di vista dell'espressione, questa risulta estremamente semplice in quanto già la stessa descrizione ci dice come dev'essere l'espressione

logica: in questo caso u0=s' enable; u1= s enable.

Quando può essere utile utilizzare un decodificatore: ad esempio, immaginiamo di avere due moduli che chiamiamo m0 e m1 (potrebbero rappresentare in particolare due moduli di memoria) che non possono essere attivi contemporaneamente, solo uno dei due deve essere attivo, l'altro no. Supponiamo che questi due moduli debbano pilotare la stessa linea e che le uscite dei due stessi moduli siano collegate direttamente alla stessa linea, succede che solo uno dei due moduli manda in uscita su questa linea il contenuto di una certa locazione di memoria l'altra no. Per fare in modo che solo uno dei due moduli mandi sulla linea un valore bisogna che arrivi ai due moduli un segnale di abilitazione o non abilitazione, questo segnale potrebbe essere ad esempio l'uscita di un decodificatore. Il segnale che arriva potrebbe avere valore 0 o 1: se l'enable è attivo e il segnale di abilitazione s d'ingresso del decodificatore è 0 allora viene abilitato il modulo m0 per inviare il dato sulla linea, se s=1 allora è abilitato m1 ad inviare il dato sulla linea, se l'enable è disattivato qualunque sia il valore di s nessuno dei due moduli è abilitato e quindi nessuno dei due può invaire dati sulla linea. Quando il modulo non è abilitato potrebbe accadere che la linea viene messa in alta impedenza, cioè è come se la linea diventa aperta (comportamento da circuito aperto) e non vi fosse un collegamento, è come se i moduli fossero staccati dalla linea e quindi non possono pilotarla. Il decodificatore serve quindi ad abilitare un oggetto piuttosto che un altro; non potrà mai capitare che entrambe le linee siano contemporaneamente a 1 pechè vorrebbe dire che entrambi i moduli sono abilitati a mandare dati sulla linea e in questo caso si avrebbero situazioni di indeterminatezza.

Caso di decodificatore con 4 uscite (2x4): in questo caso, avendo sempre l'enable presente, succede che solo una delle 4 uscite quando l'enable è attivo vale 1 tutte le altre saranno a 0. In questo caso pensando alla linea q0 questa è 1 solo se l'enable vale 1, s1=0 e s0=0; altrimenti q0=0. Poichè le espressioni logiche del decodificatore considerato risultano estremamente semplici da qui ne segue che andare a fare la sintesi del decodificatore è conveniente.

Supponiamo ora di voler costruire un enable 2x4 a partire da due 1x2: in questo caso è meno immediato il procedimento rispetto a quello del mux, perchè come risultato finale dovremmo avere i valori di q3, q2,q1 e q0. Se andiamo a guardare s1 e s0 notiamo che: l'uscita u=1 se l'enable=1 e poi abbiamo 0,0. Q1 quando l'enable vale 1 e s1=0 e 1. In questo caso le uscite sono 1 se s1=0; invece le uscite per q2 e q3 sono 1 se s1=1. Questi risultati possono essere sfruttati per determinare i segnali d'abilitazione di questi due mux perchè se s1=1 allora potrà essere attivato o q3 o q2; se s1=0 potrà essere attivato o q1 o q0.

Un altro componente che possiamo utilizzare all'interno del sistema digitale sono i registri. Questi possono essere realizzati utilizzando in parallello tanti flip-flop quanti sono i bit che vogliamo andare a memorizzare nei registri stessi. Supponiamo di avere a disposizione un flip-flop di tipo D caratterizzato da un segnale di clock, un ingresso D e un'uscita Q. A partire da questo elemento di memoria che supponiamo avere già a disposizione possiamo costruire un registro a n bit che ha n bit in ingresso e n bit in uscita: questo è ottenuto mettendo in parallelo tanti flip-flop quanti sono i bit che vogliamo memorizzare e a ciascuno di questi flip-flop mandiamo l'ingresso corrispondente al bit da memorizzare [D(0), D(1),...D(n-1)], il clock è unico per tutti i flip-flop e le uscite saranno quelle corrispondenti a ciascun flip-flop e nell'insieme costituiranno l'uscita complessiva del registro. In questo modo attraverso il parallelo di tanti flip-flop riusciamo a costruire un registro in maniera semplice.

Vediamo adesso come realizzare un banco di registri. Tipicamente all'interno di un processore

abbiamo a disposizione dei registri di uso generale che nell'insieme possono essere letti e scritti. Le istruzioni sono caratterizzate da due operandi: per esempio, un istruzione ADD R3,R1,R2 dove R1 e R2 sono i registri sorgenti e R3 il registro destinazione, questo è interpretabile come R3=R1+R2. Questi registri saranno presenti all'interno del processore e devono essere indirizzati opportunamente, quindi devono essere letti e nel registro di destinazione dobbiamo anche poter scrivere. Il banco di registri visto come un unico oggetto possiamo caratterizzarlo con un indice che potrebbe essere per esempio il registro R1, un indice relativo al registro R2 da dove leggiamo e dopo aver letto abbiamo un DATO 1 relativo al registro R1 e un DATO 2 relativo al registro R2; in più abbiamo anche un registro R3 dove scrivere ma anche un DATO da scrivere in corrispondenza del registro R3. In più notiamo la presenza del clock per decidere quando andare a scrivere nel registro R3. Questi sono gli ingressi e le uscite del banco di registri.

Realizzare un banco di registri è un operazione un po' più complessa rispetto alla realizzazione dei singoli componenti già visti. L'approccio più semplice per la realizzazione consiste nell'utilizzare i componenti elementari già progettati per leggere e scrivere nel banco di registri. Vediamo come organizzare il banco di registri per eseguire le operazioni di lettura e scrittura: immaginiamo di avere a disposizione 4 registri su cui poter eseguire operazioni di lettura o di scrittura. Per quanto riguarda le operazioni di lettura, possiamo leggere due dei 4 registri e per quanto rigurada i dati da leggere ce li dicono R1 e R2. Per realizzare le operazioni di lettura dobbiamo fare in modo che il blocco produca due uscite ovvero quelle chiamate DATO 1 e DATO 2. Questi due dati sono l'uscita di uno dei 4 registri, quindi ci serve un mux che ci permetta dati i 4 valori dei registri di averne solo uno in uscita. Quindi, a questo punto, facciamo in modo che le uscite di tutti 4 i registri vengano mandate tutte a due mux 4x1 che producono in uscita o il DATO 1 o il DATO 2. Perchè venga mandato il valore corretto del registro dobbiamo decidere quale dei 4 registri mandare all'uno o all'altro mux; chi decide sono gli ingressi R1 e R2. L'ingresso R1 formato da 2bit mi dice quale dei 4 registri mandare in uscita e questo produrrà il DATO 1, analogamente si comporta l'ingresso R2 fatto anch'esso da due bit per il DATO 2.

Per l'operazione di scrittura : questa prevede che un dato (DATO W) venga scritto in uno dei 4 registri, questo vuol dire che solo uno dei segnali di abilitazione alla scrittura, il clock, deve essere abilitato tutti gli altri devono essere a zero perchè dobbiamo scrivere in uno solo dei registri. Il segnale che arriva ai registri è un AND tra il clock che arriva a tutti e un altro segnale che, sulla base dei 4 già presenti, deve essere l'unico pari a 1 tutti gli altri devono essere 0. Il componente che fa in modo che dei 4 valori solo uno valga 1 e tutti gli altri 0 è il decodificatore, che fa in modo di mandare l'uscita a questi AND dei 4 registri. L'ingresso, invece, del decodificatore è R3 ovvero il registro che ci dice dove scrivere. In realtà, il solo R3 nn è sufficente, abbiamo bisogno infatti di un altro segnale detto REG_WRITE che abilita la scrittura nel registro. Non sempre dobbiamo scrivere in registri, in quanto ci sono cicli di clock che non prevedono operazioni di scrittura; quindi se non dobbiamo scrivere qualunque sia il valore presente in ingresso non deve essere possibile la scrittura nei 4 registri quindi tutte e 4 le uscite devono essere uguali a 0. Se non scriviamo REG_WRITE=0 e quindi la scrittura è disabilitata, quando invece la scrittura è abilitata allora REG_WRITE=1 e viene posta a 1 uno delle 4 uscite in base al valore di R3. questa struttura può essere estesa anche ala caso di più registri, in questo caso l'unica cosa che cambia sono le dimensioni del multiplexer e del decodificatore.

slide : comparatore a n bit (A==B)

Il comparatore a n bit può essere realizzato a partire da quello ad un bit. Se dobbiamo realizzare un comparatore a n bit basterà prendere n comparatori ad un bit. Il risultato sarà un vettore di n bit relativo ad ogni coppia di bit: a questo punto basta fare un AND tra tutti i risultati ottenuti per ogni comparatore a 1bit. Il risultato dell'AND ci dirà se i due vettori sono uguali oppure no. I vettori devono però avere la stessa dimensione se no il confronto non è possibile.

Caso di un sommatore: col termine full-adder si indica un sommatore ad 1bit, esso è caratterizzato dagli ingressi A,B e C in e produce S e Cout. Possiamo utilizzare i sommatori a singolo bit per realizzarne uno a n bit: possiamo ottenere questo risultato in due modi:

1. con la propagazione di riporto in cui mettiamo in cascata tanti sommatori quanti sono i bit da sommare;

2. con l'anticipo di riporto.

Lezione 20 aprile 2011

Con l'anticipo di riporto i vari hardware ricevono un carry che sostanzialmente viene prodotto localmente utilizzando questa formuletta in cui il carry di indice i+1 viene ricavato come la somma di due termini: Gi+PiCi = aibi+ai(xor)biCi. Facendo un esempio: Ci è il carry generato dal bit 0 e usato dal bit 1. La formula è espandibile ricorsivamente…. La complessità del circuito aumenta all'aumentare del numero dei carry.

Pag16->Realizzazione fisica dei sommatori
Nel caso del bit 0 il carry in ingresso è proveniente dall'esterno che viene utilizzato per essere combinato
con a0 e b0 per produrre la somma s0 e un carry c1; il carry 1 può essere calcolato da un blocco CarryLookahead che fa l'anticipo del calcolo del carry.
Il secondo blocco esegue la somma di a1 e b1 utilizzando il carry 1 che è stato calcolato combinando a0 b0
e c0 dal CarryLookahead, e così via…
Man mano che i bit in ingresso aumentano, anche la complessità per il calcolo del carry aumenta eccessivamente.
Spesso per diminuirne la complessità si sceglie di calcolare il carry usando un sottoinsieme di bit sui quali
eseguirne il calcolo con anticipo di riporto limitando il numero di bit in ingresso al CarryLookahead.
Pag17-> Esempio
Come esempio si può vedere che il carry d'uscita del blocco 2 viene mandato direttamente nell'ingresso
dell'altro full Adder; da questo momento in poi si ricomincia da capo nel calcolo del carry con anticipo di
riporto. Con una soluzione di questo tipo potremmo avere un tempo di ritardo maggiore ma un costo realizzativo di certo più contenuto rispetto all'approccio con anticipo di riporto puro in cui, il calcolo del
riporto degli ultimi bit può essere molto oneroso dal punto di vista dell'area occupata.
Dovendo trovare un compromesso tra velocità e area la soluzione consiste nel dover interrompere la procedura e ricominciare da capo.
Uno dei parametri di progetto potrebbe essere quello di stabilire quanti blocchi utilizzare per l'anticipo di
riporto; tanto maggiore il numero di tali blocchi, quanto minore sarà il ritardo (Il costo aumenterà proporzionalmente).

Pag18-> Differenza
Se volessimo eseguire la differenza utilizzando l'adder, evitando così di dover sintetizzare un ulteriore
dispositivo, si potrebbe fare in modo che l'ingresso B venga complementato, e mandato insieme al suo
valore originario al multiplexer che provvederà ad inoltrare di volta in volta la variabile richiesta nel secondo ingresso dell'adder (dipendentemente dal fatto che si voglia eseguire la somma o la differenza).
In sintesi l'adder riceverà la variabile A nel primo ingresso, e la variabile B (o b negato) al secondo ingresso
con un carry che sarà pari a 0 nel caso della somma ed a 1 nel caso della differenza…
In questo modo si riesce ad ottenere anche la differenza sfruttando la rappresentazione in complemento a
due.

Pag19-> Schema molto semplice di come potrebbe essere rappresentata un'ALU.
Una rappresentazione semplice consiste nel progettare un componente per ogni azione che l'ALU debba
eseguire. In questo esempio l'ALU a due ingressi (A e B) a n bit che vengono mandati in parallelo a tutti i
blocchi ciascuno dei quali fa una delle operazioni che l'ALU è in grado di eseguire: Sommatore, AND bit a
bit, OR bit a bit, NOT(in questo caso relativo ad A), SLT(Set on Less Then) confronto di minoranza(if(a<b)
ris=1; else ris=0;), Set on Less Equal (sta per minore o uguale), Set Equal (sta per Uguale)(N.B. Sono
istruzioni di confronto che verranno realizzate con dei comparatori), SLL (Shift Left Logical) e SRL (Shift Right
Logical) che non fanno altro che ruotare bit all'interno di un registro.
N.B. Il bit meno significativo verrà posto uguale a 0.(Esiste anche una versione di Shift di tipo circolare)
Ogni spostamento verso sinistra comporta una moltiplicazione per 2.(n spostamenti comportano una moltiplicazione per 2^n). In modo complementare uno spostamento verso destra comporta una divisione
per 2. Lo Shift è utile per eseguire operazioni di moltiplicazione e divisione senza ricorrere all'utilizzo di
moltiplicatori e divisori che richiedono un costo e, un tempo di ritardo(latenza) di gran lunga maggiori.
Gli Shift richiedono numerosi cicli di clock a differenza di moltiplicatori che sono più veloci ma occupano
un'area maggiore. La scelta di uno Shift o di un moltiplicatore è dovuta alla necessità di avere un
processore che sia o più veloce a discapito dell'area occupata(quando non si hanno problemi di spazio), o
più piccolo e più lento (se si hanno restrizioni riguardo l'area occupabile): è necessario arrivare ad un
compromesso. Le uscite dei vari componenti verranno mandate ad un unico multiplexer che, a sua volta,
produrrà un'unica uscita coincidente con il risultato generato dal componente a cui si fa riferimento, mediante il segnale op (a 3 bit).
Questa è un'ALU semplice ma non efficiente in quanto tutti i componenti sono attivi eseguendo ciascuno la
propria operazione; il multiplexer si occuperà di mandare in uscita il risultato corretto.

Quest'implementazione inoltre, è notevolmente dispendiosa dal punto di vista della potenza. Si può fare in
modo che gli ingressi dei vari componenti non cambino (evitando così che si attivino),se non necessario,
risparmiando così potenza dinamica. (Si distingue tra potenza Statica e Dinamica: vedi Elettronica).

Progettazione di un Sistema digitale
La rete di un sistema digitale è composta da due parti: il Datapath che esegue le azioni richieste e produce
un certo risultato e L'unità di Controllo che genera i comandi per il Datapath attraverso dei segnali(vettore
di bit).
Il Datapath è costituito da tutti quei componenti necessari per realizzare le operazioni che il sistema digitale
deve eseguire: esso sarà composto da reti combinatorie (sommatori, multiplexer,comparatori…) e anche da
possibili registri di uso generale o specifici.
L'unità di controllo invece, è possibile immaginarla come una macchina a stati (rete sequenziale sincrona)
che partendo dalla stato iniziale, evolve nel tempo ed in base allo stato in cui si trova (condizioni fornite dal
Datapath) producendo i corretti segnali(comandi) per il Datapath.

Supponiamo di dover calcolare il minimo comune multiplo.
Quando arriva il segnale Start(attivo) ha inizio il calcolo del mcm tra A e B in ingresso. Non appena il
risultato è disponibile il sistema deve produrre tale risultato e deve attivare il segnale Ready che dimostra
che il risultato è pronto: fintanto che il segnale Ready non è stato attivato il risultato in uscita non coinciderà con il mcm.
Non appena Start=1, Ready sarà posto a 0 e ritornerà ad 1 solo quando il calcolo sarà terminato.
Il sistema pertanto è composto dai tre ingressi A, B, e Start (con in più il clock in quanto si tratta di rete
sincrona) e produce in uscita il risultato e il segnale Ready.
Si presuppone che A e B siano due numeri maggiori di 0 affinchè tale algoritmo funzioni. Una volta
realizzato l'algoritmo si può risalire al numero di componenti necessari. Saranno necessari uno o due
sommatori (è possibile nel caso più semplice scegliere due sommatori distinti, utilizzando più hardware ma
riducendo i comandi dell'unità di controllo, per eseguire la somma di ma+A ed mb+B o poiché le due
operazioni andranno eseguite in momenti differenti, si può decidere di utilizzare un unico sommatore
complicando lo schematico), due o più comparatori, multiplexer (in quanto la variabile ma (come mb) varia
il suo valore da A ad ma+A), registri (che dovranno contenere di volta in volta i valori aggiornati delle
variabili) o in sostituzione delle locazioni di memoria (gli aggiornamenti non devono avvenire sempre ma
solo in opportuni cicli di clock per cui l'aggiornamento deve essere cadenzato dal clock). Il clock deve
rimanere attivo solo al momento dell'aggiornamento.
Il passo successivo consta nel dover fare la sintesi dei vari componenti necessari. Tali componenti

dovranno
poi essere opportunamente combinati in modo da realizzare il Datapath. L'unità di controllo provvederà di
volta in volta a inviare i giusti segnali al Datapath (SelA, SelB, WriteA, WriteB….). Il Datapath produrrà i
segnali diverso e minore.
Realizzato il Datapath si procede con la sintesi della Control Unit utilizzando i metodi classici di sintesi di reti
sequenziali; l'unico problema è la realizzazione della macchina a stati. Dell'unità di controllo bisogna
stabilire quali sono gli stati prossimi e i valori dei segnali write A, write B… Come macchina a stati si può
scegliere la macchina di Moore in cui le uscite dipendono unicamente dallo stato. Come stato iniziale
potremmo immaginare uno stato in cui il sistema non fa nulla fino a quando non arriva un segnale di Start
che abilita l'inizio del calcolo; quando Start=1 andiamo in uno stato d'inizializzazione dei registri di ma e mb.
A questo punto potremmo spostarci in uno stato di confronto, in cui verranno confrontati ma e mb per
stabilire quale azione intraprendere: se ma<mb ed ma!=mb -> ma=ma+A e si ritorna a fare il confronto; se
ma>mb ed ma!=mb -> mb=mb+B e si ritorna a fare il confronto; se ma=mb -> ris=ma e si ritorna allo stato
di Idle. N.B. l'unità di controllo decide in base ai segnali emanati dal Datapath.
Dalla tabella degli stati di Pag11 (slide progettazione sistema digitale) è possibile visualizzare le variazioni
che subiscono i segnali.
Nei casi più complessi vengono usati dei tool che basandosi sui vari segnali sintetizzano l'unità di controllo.
Opportuni linguaggi orientati all'hardware permettono una sistesi di più alto livello del sistema digitale.

Architettura dei calcolatori:
unità di elaborazione=Datapath
La differenza tra il processore di un calcolatore elettronico e il sistema digitale precedentemente studiato
consiste nel fatto che mentre nel sistema digitale siamo in grado di eseguire un unico programma (minimo
comune multiplo) ovvero si tratta di un processore Single Purpouse (il programma è implicito nella
struttura stessa del Datapath),in un calcolatore elettronico è possibile eseguire più programmi suddivisi in
istruzioni (opportunamente salvate in memoria), che vengono sequenzialmente richiamate dalla memoria
ed interpretate dal Control Unit (decodifica delle istruzioni) che genera gli opportuni segnali di controllo per
il Datapath, e che a sua volta realizza le istruzioni previste. Completata l'esecuzione, il ciclo ricomincia
(Lettura -> decodifica -> Esecuzione). I componenti tipici di un calcolatore di controllo sono: il processore
suddiviso in Datapath e Control Unit (parte attiva); memoria (parte passiva); dispositivi I/O.

Pag4-> Possibile collegamento dei vari dispositivi

Il controllore di canale non è altro che il DMA(Direct Memory Access)
Il collegamento tra memoria e CPU e detto bus di sistema

Pag5-> Altra alternativa di collegamento: tutti i dispositivi sono collegati mediante un unico bus
I vantaggi di questo approccio consistono nei costi più contenuti rispetto all'approccio precedente; lo svantaggio consiste nel fatto che è possibile eseguire una comunicazione per volta poiché il bus è unico: ciò comporta un rallentamento del sistema.

Pag6-> Struttura generica dell'organizzazione di un pc corrente
Come si può notare in un recente Personal Computer, sono presenti numerosi bus che permettono il collegamento con tutte le risorse del sistema e vengono gestiti da dei blocchi indicati con il termine Bridge.
Un bridge è un componente che permette la comunicazione tra un bus ed un altro. La complessità (al giorno d'oggi) è molto più elevata ed è necessario l'uso dei Bridge per permettere comunicazioni tra i vari dispositivi, passando tra più bus. Il passaggio da un bus a un altro può causare dei conflitti nel caso in cui l'accesso al bus è conteso tra più dispositivi; bisognerà pertanto definire dei meccanismi che stabiliscano delle priorità sull'accesso ai bus.

Pag7-> Il nostro schema di riferimento sarà corrispondente allo schema usato dai PC negli anni 80, basato su di un unico bus di sistema che permetta il collegamento tra la CPU, la memoria e le periferiche di I/O.
Tale bus di sistema risulterà al suo interno diviso in tre parti: il bus dati; il bus di controllo(segnali necessari per la comunicazione); il bus degli indirizzi (riferiti alla memoria o ai dispositivi di I/O identificati attraverso un unico indirizzo).

Pag8-> Diagramma temporale che rappresenta il comportamento dei vari segnali duranti il ciclo di lettura.
M segnato e I/O vanno così interpretati: 0 logico corrisponde ad un uso della memoria mentre 1 Logico all'uso di una periferica di I/O. Allo stesso modo: Read segnato indica che la lettura avviene in presenza di 0 Logico e Write segnato indica che la scrittura avviene in presenza di 0 Logico. Immaginando che la lettura abbia inizio sul primo fronte di clock, dopo un breve periodo di transitorio apparirà sul bus indirizzi un indirizzo valido; questo indirizzo sarà mandato alla memoria (specificato da M-I/O) e verrà eseguita la lettura (specificata dalla coppia READ-WRITE); sul bus di dati sarà presente inizialmente un valore non valido (residuo di qualche procedura precente) e solo dopo un certo intervallo di tempo (che dipende dalla velocità della memoria) tale valore verrà aggiornato (ovvero risulterà valido) e verrà letto dalla CPU in corrispondenza del fronte (nel nostro caso sul fronte di salita in quanto la lettura ha avuto inizio sul

primo
fronte di salita). N.B. Si presuppone che la CPU conosca il numero di cicli di clock che deve attendere prima
di poter leggere sul bus dati; nel caso in cui questo ritardo non sia noto a priori, può essere necessario
aggiungere un ulteriore segnale (ad esempio Ready) prodotto dalla memoria una volta prodotto il dato.
Perché la CPU possa leggere i valori correttamente, è necessario che questi permangano sul bus di dati per
un certo intervallo di tempo (noto come tempo di Hold: caratteristica dei registri della CPU) anche dopo il
fronte di lettura del clock.

Pag9-> Ciclo di Scrittura: simmetrico alla lettura.
In questo caso il bus dei dati è subito disponibile mentre il problema sorge sulla durata della scrittura: tale
durata dipende sempre dal tipo di memoria e dalla sua velocità, che la CPU può conoscere a priori, o può
essere necessario l'utilizzo di un ulteriore segnale (mandato dalla memoria) che provveda a segnalare alla
CPU quando il ciclo di scrittura è terminato.

Pag10->Von Neuman
In questa architettura la CPU è collegata ad un'unica memoria che contiene sia i dati che le istruzioni;
questa configurazione è poco costosa ma risulta essere problematica in quanto non permette la
realizzazione di più istruzioni contemporaneamente (in parallelo).

Pag12-> Architettura di Harward
Nel caso in cui è necessario eseguire più operazioni in contemporanea (lettura di dati e istruzioni) sono
necessarie più memorie (una per i dati e una per le istruzioni) che risultano essere inoltre differenti in
dimensioni e velocità. Quest'approccio è molto usato nei sistemi DSP in quanto il numero di istruzioni da
eseguire risulta essere molto elevato. Il raddoppio dei bus, dei pin della CPU e della memoria implica un
aumento del costo ma anche un aumento significativo delle prestazioni in termini di tempo.

Lezione 27 Aprile 2011
Pag13-> Struttura CPU
Il processore studiato sarà strettamente sequenziale ovvero alla fine di un'istruzione se ne esegue un'altra.
Vi sono processori capaci di eseguire più istruzioni in parallelo: ciò causa una modifica nell'organizzazione
del Datapath. Il Datapath dipende dalle istruzioni eseguibili dalla CPU e dall'organizzazione che noi
vogliamo darle. L'unità di controllo dovrà fornire i giusti comandi al Datapath: segnali per l'abilitazione di
multiplexer, decodificatori,… questi comandi dipendono dall'istruzione da eseguire e da eventuali
condizioni emanate dal Datapath stesso in base ai risultati dell'elaborazione. L'esecuzione di un'istruzione
può essere suddivisa in due macro fasi: Fetch (con decodifica) ed Execute (che può essere più o meno breve

in base al tipo di istruzione).

Pag16-> Elementi di una CPU (da non confondere con lo schematico che fornisce info anche sui collegamenti).
L'unità di controllo è formata da due parti: una logica di controllo che fornisce i vari comandi che devono essere inviati al Datapath, e l'IR (Instruction Register) che contiene l'istruzione in corso di esecuzione e che permane fino alla lettura dell'istruzione successiva. Il Datapath è composto dall'ALU che svolge tutte le operazioni logico-aritmetiche, eventuali registri di uso generale (in numero limitato: da 32 a 64) che contengono i valori delle variabili in uso per le varie istruzioni (avere le variabili mappate direttamente sui registri migliora la velocità di esecuzione riducendo il numero delle istruzioni*), e da registri per uso specifico tra i quali: PC (Program Counter) che tiene traccia dell'istruzione successiva e pertanto verrà incrementato di volta in volta di un certo numero di byte (tale numero dipende dal tipo di istruzioni); MAR (Memory Address Register) contiene l'indirizzo della memoria da mandare nel bus degli indirizzi (sul bus degli indirizzi viene inviato il contenuto del PC per passare all'istruzione successiva, o l'indirizzo del MAR per leggere o scrivere da memoria); MDR (Memory Data Register) registro attraverso il quale passano i dati da leggere o scrivere dalla memoria.

*a causa del numero limitato dei registri solo un sottoinsieme di variabili può essere mappato in tali registri mentre le altre variabili vanno salvate in apposite locazioni di memoria: ciò comporta un maggiore numero di istruzioni (tra le quali load e store) ed una velocità di esecuzione notevolmente ridotta (la fase di Execute è pertanto più lunga).

Pag19-> Descrizione semplificata della fase di Fetch ed Execute.
N.B. il PC viene incrementato alla fine della fase di Fetch non in quella di Execute…
Nella fase di Execute viene eseguita una delle istruzioni sostenibili dal processore; per semplicità si è supposto che la fase di Execute sia unica, mentre è possibile che sia composta da più sotto-fasi.

Pag20-> Struttura semplificata della CPU.
PSW (Program Status Word) è un registro di stato che contiene alcuni dei risultati che si ottengono durante l'esecuzione.

Pag21-22-23-24-25-26-27-> Lettura dalla memoria.
Immaginando che le operazioni avvengano in un unico ciclo di clock, sono evidenziate in rosso le periferiche utilizzate durante la fase di Fetch (Decode) ed Execute secondo quanto è stato detto (Pag19)…

Pag28-> Nel caso di somma tra due numeri non è necessario accedere alla memoria in quanto i dati sono

salvati nei registri, per cui la fase di Execute in questo caso richiede un unico ciclo di clock per leggere dai
registri, eseguire l'operazione e salvare il risultato in un terzo registro.

Differenti Implementazioni del processore…
Pag29-30-31-> Nel caso in cui il numero di bus sia ridotto, non è possibile eseguire le operazioni di lettura e
scrittura dai registri in un unico ciclo di clock pertanto per eseguire le stesse operazioni fatte in precedenza
saranno necessari più cicli di clock(Lettura da R1, Lettura da R2, Salvataggio in R3: in totale necessitano 3
cicli di clock). Il numero di cicli di clock non dipende solo dal tipo di istruzione ma anche dall'organizzazione
scelta per implementare l'istruzione.

Pag32-> In questo caso sono presenti tre bus diversi, pertanto è possibile eseguire lettura e scrittura da e
verso i registri in un unico ciclo di clock.

Pag33-> In questa terza versione sono presenti due soli bus; sono stati aggiunti dei registri e si può notare
la presenza dei segnali SCL e SCS che controllano la scrittura e la lettura, ed il segnale SCop che fornisce il
giusto comando all'ALU. Inoltre è presente un "blocchettino" triangolare detto buffer tri-state (regolato dal
segnale SCL) che permette di evitare che più registri accedano allo stesso bus contemporaneamente
causando incongruenze (Esso si comporta da interruttore). Un'alternativa a questo buffer potrebbe essere
l'utilizzo di un multiplexer che però ha lo svantaggio di coprire un'area maggiore (anche il costo è più
elevato).
N.B. I segnali vanno abilitati di volta in volta solo quando è necessario; tutti gli altri devono essere disabilitati.

Pag36-> Possibile implementazione di un Istruction Set che ci premetterà di realizzare il Datapath e la
relativa unità di controllo. Il set d'istruzioni che stiamo analizzando e simile a quello di un processore MIPS
di tipo RISC ovvero con un set d'istruzioni ridotto ma estremamente veloci. Istruzioni di tipo R ovvero che
coinvolgono solo registri come ad esempio l'istruzione add rd, rs, rt -> Reg[rd] = Reg[rs]+Reg[rt] ovvero il
contenuto del registro alla posizione rd deve essere uguale alla somma dei contenuti del registro alle
posizioni rs ed rt. N.B. Ogni istruzione occupa 32 bit: i primi 6 bit sono detti Codice Operativo e stanno ad
indicare il tipo di operazione da eseguire; i seguenti tre gruppi di 5 bit sono occupati dai registri a cui si fa
riferimento; dal 21 al 25 bit non sono utilizzati; gli ultimi 6 bit il tipo di operazione da eseguire: nel nostro
caso la somma.
Istruzioni di Salto che permettono di saltare (in avanti o indietro) da un operazione ad un'altra. Quello
mostrato rappresenta una tipologia di salto condizionato (Branch Equal) che confronta il contenuto

del
registro rs con il contenuto del registro rt: if(Reg[rs]==Reg[rt])
PC=PC+estensione_in_segno(spiazzamento<<2)<-(Shift a sinistra)
Lo Spiazzamento è un numero in complemento a due che può essere positivo o negativo.

Pag37-> Load Word serve per caricare valori dalla memoria; Store Word viene usato per memorizzare valori
in memoria. Si può notare come sia presente una certa regolarità nella suddivisione dei bit che rende tale
struttura rigida e veloce; una struttura più variabile comporta una decodifica più lenta.

Pag38-> Per poter realizzare correttamente un Datapath bisogna studiare quali componenti sono necessari
per poter eseguire le varie istruzioni della CPU. In questo caso si suppone che nella fase di Fetch non siano
necessari i registri MAR e MDR per rendere più semplice e veloce l'operazione(IR:=M*PC+; PC:=PC+4)
(supponendo che le istruzioni occupano 4byte). I componenti necessari pertanto saranno il PC, l'ALU, l'IR e
la memoria che non fa parte del Datapath ma è utile per evidenziare i segnali di controllo necessari ad
utilizzarla.

Pag39-> Nella fase di Decodifica delle istruzioni saranno necessari: l'IR avente al suo interno il valore
inserito nella fase di Fetch; i registri A e B in cui verranno memorizzati gli indirizzi rs ed rt (questo passaggio
viene eseguito sempre anche nel caso in cui non serve); un Sommatore che salva la somma tra il PC e lo
Shift di 2 a sinistra dell'estensione in segno del lo spiazzamento, salvando il risultato in un registro chiamato
Target: tale valore coincide con l'indirizzo di destinazione in caso di salto condizionato. (Quest'operazione
va eseguita sempre anche nel caso in cui non si tratti di un'istruzione di Branch poiché eseguita in parallelo
con la decodifica, non rallenta il sistema).

Pag40-> Nella fase di Execute, se si tratta d'istruzione di tipo R, sarà necessario l'IR, i registri A, B e
ALUOutput, e l'ALU.

Pag41-> Nel caso in cui l'Execute comporti un'istruzione di Branch invece saranno necessari anche il Target
contenente l'indirizzo calcolato nella fase di Decode ed il PC che andrà aggiornato con tale valore.
ERRORE: l'OR va sostituito con un AND.

Pag42-> Nel caso in cui l'Execute comporti accesso a memoria (load o store)necessitano: IR, A, ALUOutput,
ALU ed il componente per l'estensione in segno.

Pag43-44-> Evidenziano i casi di lettura e scrittura in memoria che comporta l'uso di due nuovi registri.

Pag45-46-> Mostrano come in caso d'istruzioni di tipo R o di letture in memoria, il contenuto sia memorizzato nei registri mediante un processo di scrittura denominato Write Back.

Pag47-> Riunendo tutti i componenti visti finora utilizzando opportuni Multiplexer si ottiene l'unità operativa in figura. In questa figura mancano alcuni segnali di controllo, tra i quali quelli che autorizzano la
scrittura in A, B, Mem Data ed AluOutput.

Pag48-49-50-51-> È evidenziato in blu quali componenti vengono coinvolti nelle tre fasi (Fetch, Decode ed
Execute) secondo quanto già affrontato nel dettaglio. I numeri presenti in prossimità dei componenti coinvolti indicano il Codice Operativo che viene loro inviato.

Pag52-> Riassunto delle istruzioni di tipo R

Pag54-55-56-> È evidenziato quali componenti vengono utilizzati in caso di Execute se viene richiesto
l'accesso in memoria in lettura. Le fasi di Fetch e Decode rimangono immutate.

Pag57-> Riassunto delle operazioni con accesso in memoria in lettura.

Pag58-> Fase di Execute in caso di Branch

Pag59-> Riassunto delle istruzioni di Branch

Pag60-61-> È evidenziato quali componenti vengono utilizzati in caso di Execute se viene richiesto l'accesso
in memoria in scrittura.

Pag62-> Riassunto delle operazioni con accesso in memoria in scrittura.

Pag63-> Macchina a stati del processore, dove sono schematizzate tutte le operazioni eseguite dal processore stesso in caso di istruzioni di tipo R, Branch, ed Accesso in Memoria. Si nota facilmente che la
fase di Fetch e Decode rimane la stessa, qualsiasi sia la fase di Execute.

Pag64-65-66-> sono illustrati i vari sottoinsiemi dei segnali di controllo che vanno attivati nelle varie fasi
dell'operazione, secondo quando illustrato negli schematici precedenti.

Il passo successivo consiste nella realizzazione dell'unità di controllo che può avvenire o con le tecniche
classiche di sintesi di reti sequenziali che sarebbe molto onerosa in caso di un numero eccessivo di stati (per
non dire impossibile quando il numero di stati supera il centinaio o il migliaio), oppure si utilizzano dei
software o dei linguaggi specifici per la sintesi.

Lezione 2 maggio 2011

abbiamo visto come progettare il Data path e quindi come costruire la macchina a stati di quest'ultimo adesso andiamo a realizzare l'unita di controllo, abbiamo due possibilità per fare l'unita di controllo: il primo approccio e la realizzazione cablata dell'unita di controllo tramite le reti sequenziali formata da registri che memorizzano lo stato in cui si può trovare la CPU e una rete

combinatoria che in base allo stato produce lo stato prossimo e i segnali di controllo per il datapath abbiamo diverse possibilità di rete combinatoria: le uscite dipendo dal valore del registro di stato e dagli ingressi la complessità della rete combinatoria la calcoliamo quando quest'ultima è usata per formare una memoria ROM costituita da un certo numero di locazioni che sono dei vettori che è l'insieme dei segnali di uscita per l'unita operativa e per lo stato prossimo. La dimensione della memoria ROM e data da numero di uscite per unita operative + (log(base 2) del numero degli stati) il tutto * 2^numero di ingressi , questo se rappresentiamo la rete come macchina di mealy, ma con questo approccio si occupa area inutile perché ci saranno parti della memoria inutilizzate o utilizzate molto di rado, area che può essere usata per ingrandire il Data Path, invece se rappresentiamo il tutto con una macchina di moore la dimensione diventa 2^numero di ingressi*(log(base 2)Del numero degli stati)+numero stati * le uscite, una soluzione migliore perché qui non c'è memoria inutilizzata.
NB: per codificare ad es 9 stati ci vogliono 4 bit
Questo Lavoro di progettazione non può essere fatto a mano richiederebbe troppo tempo costruire tabelle della verità ecc.... Oggi Grazie ad Alcuni tool il lavoro di progettazione è notevolmente semplificato, infatti noi andiamo ad descrivere il funzionamento con linguaggi di alto livello senza specificare i singoli segnali di controllo, ovvero linguaggi Hw di descrizione simili hai linguaggi di programmazione. I tool fanno una traduzione del nostro linguaggio e ci danno come risultato una descrizione a basso livello tramite componenti elementari come sommatori,registri, MUX,DEC ecc.... un linguaggi esempio e il DHDL Quindi oggi siamo in grado grazie a questi strumenti di progettare strutture molto complesse con migliaia di porte logiche e milioni di transistor, progettare sistemi così complessi a mano ci porterebbe via decine di anni, i primi sistemi a nascere naturalmente avevano complessità molto basse e quindi potevano essere progettati a mano.
La seconda possibilità per progettare l'unita di controllo è usare la logica programmabile dove a livello Hw quello che si ottiene è una struttura formata dai diversi dispositivi (es porte logiche) che sono collegati tra di loro elettricamente per ottenere la funzionalità specificata dall'istruzione che è stata appena eseguita. Quindi in questo tipo di sistema non si devono inserire gli elementi uno ad uno e collegarli in modo opportuno per ottenere la funzione richiesta, ma gli elementi sono già presenti e bisogna soltanto programmarli (in genere tramite interruttori) per ottenere la funzione richiesta. La progettazione Di questo tipo di sistemi richiede lo stesso tempo di quelli a logica Cablata, anche qui possiamo aiutarci nella progettazione tramite l'uso di tool specifici.
Un altro approccio è quello microprogrammato usato nella famiglia x86 fino a qualche anno fa, dove ogni unita di controllo è come se al suo interno ne contenesse un altra, qui abbiamo una memoria di microprogramma al cui interno sono presenti alcune Microistruzioni che a sua volta saranno composte da microoperazioni queste microoperazioni potrebbero coincidere nel caso più estremo con i singoli segnali da mandare al Data Path (in questo caso non serve il decodificatore), tutte le microoperazioni vengono in genere eseguite in parallelo, se abbiamo due microoperazioni che non possono essere eseguite in parallelo allora non possono essere contenute all'interno della stessa microistruzione, ogni microoperazione indica all'unita operativa cosa deve fare, potrebbero esserci casi in cui ad ogni microoperazione non coincide un solo segnale in questo caso serve un decodificatore che per ogni microoperazione ci da fuori i segnali di controllo relativi, una microistruzione è un vettore al cui interno sono presenti le microoperazioni, ed ad ogni microoperazione può corrispondere un certo set di segnali da inviare al data path, Se un segnale è attivo dal decodificatore uscirà un 1, altrimenti uscirà uno 0 se ad esempio possono entrare dieci segnali nel decodificatore ed ad una microoperazione ne sono associati tre ad esempio il primo il quinto ed il settimo, uscirà dal decodificatore 1000101000 ecc.... il Micro Program Counter PC contiene l'indirizzo della prossima microistruzione da eseguire, il PC viene anche mandato ad un sommatore quando viene letta l'istruzione al suo interno, cosi che il suo valore venga aggiornato con l'istruzione successiva, il selettore dell'indirizzo invece serve per effettuare il salto da una microistruzione ad un altra non consecutiva, la memoria di microprogramma coincide con la memoria ROM della realizzazione cablata, ogni istruzione data alla CPU ha sempre un ciclo di Fetch, Decode, Execute naturalmente fetch e decode sono sempre le stesse qualsiasi sia l'istruzione, l'Execute invece no perché può essere di tipo ad es R,LW ecc.... (R=op sui registri, LW=lettura

scrittura in memoria), dopo la fase di Execute qualsiasi essa sia si ritorna sempre alla fase di Fetch che ricordo è uguale per tutte le istruzioni. La complessità di queste CPU è inferiore a quella delle precedenti, l'unica difficoltà sta nel costruire la memoria di microprogramma, l'unica anomalia di questo tipo di approccio è che possiamo avere memorie di microprogramma molto grandi, tutto dipende dal numero di microoperazioni inserite nella nostra memoria di microprogramma, se abbiamo un grande parallelismo, abbiamo grandi possibilità prestazionali potendo eseguire molto microoperazioni in parallelo ma nel momento in cui ne dobbiamo eseguire poche sprechiamo una grossa quantità di memoria. Un altro lato che potrebbe essere non vantaggioso di questo approccio è che se dobbiamo essere noi ad andare a definire le microistruzioni, non otteniamo alcun vantaggio rispetto all'approccio cablato, ecco perché esistono dei linguaggi per programmare queste memorie di microprogramma in modo più semplice è comprensibile.
Abbiamo due approcci di microprogrammazione:

- microprogrammazione orizzontale
- microprogrammazione verticale

In quella orizzontale si cerca di eseguire il massimo numero di microoperazioni in parallelo, un caso estremo e quando ci sono un numero di microoperazioni pari al numero di segnali possibili, questo approccio richiede una dimensione della memoria molto elevata, e visto che non sempre tutti i segnali sono attivi nello stesso momento è inutile avere memorie enormi inutilizzate. Il vantaggio di questo approccio e quello che avendo il massimo parallelismo possibile andiamo al massimo della velocità.
In quello verticale abbiamo più microistruzioni perché ognuna è composta da meno microoperazioni quindi un parallelismo inferiore, quindi una velocità di esecuzione più lenta, ma abbiamo uno spreco di memoria inferiore.
In generale se aumentiamo l'area (quindi la memoria) miglioriamo le prestazioni, se invece vogliamo che l'area occupata dalla nostra CPU sia inferiore peggioriamo le prestazioni, quindi dobbiamo cercare un buon compromesso tra le due cose.
Facciamo un excursus storico:
Fino a fine anni ‘60: logica cablata

• Anni ‘70: microprogrammazione

– Repertorio di istruzioni molto esteso e variato: CISC

– Il VAX 11/789 (Digitale) e il 370/168 (IBM) avevano oltre

400.000 bit di memoria di controllo

• Dagli anni ‘80 si è tornati alla logica cablata;

– Affermazione delle macchine RISC

• Istruttivo è esaminare l’evoluzione dell’architettura Intel: da

CISC a (praticamente) RISC

Inizialmente avevamo CPU a logica cablata, ma con il passare del tempo si ci rese conto che programmare con questo tipo di CPU era difficile per i programmatori perché non esistevano ancora compilatori molto potenti, allora si passo alla struttura microprogrammata che semplificava il lavoro dei programmatori, perché programmare con queste CPU era più semplice. Si ritorna negli anni 80 alla logica cablata perché i compilatori sono diventati molto più potenti, anche perché molte delle istruzioni CISC non venivano mai utilizzate mentre in quelli RISC erano molto di meno le istruzioni ma molto più veloci nell'eseguire. Dagli anni Ottanta in poi anche la famiglia x86 di CPU usa un set di istruzioni RISC.

Lezione 9 maggio 2011
Se andiamo a considerare l'incremento delle prestazione dei calcolatori, si nota subito che tra le parti che crescono più rapidamente c'è la memoria dinamica o DRAM che aumenta mediamente di 1,4 volte l'anno dal 1980 al 2002 la capacita della ram è passata da 64 kb a 1 gb ovvero un aumento di 4000 volte. Adesso andiamo a considerare il numero di transistor per cip che raddoppia mediamente ogni anno e mezzo quindi le capacita di calcolo delle nostre CPU aumenta a ritmi vertiginosi.
Gli aumenti principali sono:
Processore

– Densità Logica: circa 30% per anno

– Frequenza Clock : circa 20% per anno

¼ Memoria

– Capacità DRAM capacity: circa 60% per anno

– Velocità Memoria: circa10% per anno

– Costo per bit: riduzione di circa il 25% per anno

¼ Dischi

– Capacità:

circa 60% per anno

¼ Larghezza di banda della rete:

– Aumenta di più del 100% per anno!

La frequenza del Clock fino a qualche anno fa aumentava con ritmi del 20% annuo, adesso ci siamo un pò fermati con la crescita in questa direzione, difatti adesso si ha la tendenza ad aumentare il numero di core.
Un altra cosa da notare è che la memoria aumenta del 60% l'anno come capacità, ma la nota dolente è che la velocità della memoria aumenta solo del 10% annuo.
Dalla metà degli anni 80 l'incremento delle prestazioni dei nostri calcolatori è stato circa di 1.58 x anno, mentre l'incremento tecnologico in generale è stato di circa 1.35 per anno, quindi la particolarità e che l'incremento delle prestazioni è superiore all'incremento tecnologico questo perché prima degli anni 80 si cercava solo di migliorare la tecnologia con cui venivano fatti i nostri calcolatori per aumentare le prestazioni di quest'ultimi, mentre successivamente si cominciarono ad adottare tecniche di progettazione migliori che a pari tecnologia davano prestazioni superiori, L'approccio usato nella progettazione viene chiamato Approccio Quantitativo ovvero basato su misure (in genere noi diamo importanza ad una specifica grandezza nei nostri progetti, infatti in base hai nostri scopi ci può interessare la potenza dissipata, oppure la velocità, o il costo ecc...).
Nel nostro caso si è cercato di aumentare la velocità di esecuzione delle istruzioni eseguite più di frequente, quindi evitando di spendere denaro per aumentare la velocità di esecuzione di istruzioni che vengono eseguite di rado.
Il classico esempio e quello dei processori degli anni 70 (CISC) che avevano un numero di istruzioni elevato per rendere più semplice la vita al programmatore che scriveva direttamente in

linguaggio assembly, ma se tutto questo che rende più semplice la vita al programmatore e il lato positivo, il lato negativo sta nel fatto che l'elevato numero di istruzioni (alcune molto complesse) rallenta la nostra CPU perché se guardiamo la macchina a stati della nostra CPU sarà grande è complicata questo comporta un elevato aumento della latenza sia da parte del data path che da parte dell'unità di controllo. Se al posto di avere molte istruzioni alcune complicate ne abbiamo poche ma semplici (RISC) otteniamo dei benefici nella maggioranza dei casi in cui può trovarsi la nostra CPU. Otteniamo un risultato peggiore in termini di esecuzione solo nel caso in cui dobbiamo eseguire l'istruzione eliminata come una combinazione di quelle che sono rimaste ma in genere le istruzioni complesse (quelle eliminate) sono quelle che vengono eseguite più raramente dalla nostra CPU.
Si e visto che l'80% del tempo nei processori CISC veniva usato per eseguire il 20% delle istruzioni, ecco perché quando i compilatori sono diventati più potenti (anni 80) e quindi le difficoltà del programmatore sono diminuite notevolmente perché erano questi compilatori a tradurre il codice in modo ottimale in linguaggio macchina si è passati a processori con poche istruzioni è molto veloci (RISC).
Quando costruiamo dei processori preformanti dobbiamo tenere in considerazione anche il fatto che la scrittura dei programmi per queste CPU non deve essere troppo complicata o nessuno si comprerebbe le nostre CPU, oggi non c'è più questo problema proprio perché i compilatori hanno raggiunto un certo grado di efficienza.
I motivi per cui dobbiamo calcolare le prestazioni di un calcolatore sono:

- Quantificare le caratteristiche di una macchina
- Per Capire se devo migliorare l'Hw oppure il Sw
- Per Aiutarci nella scelta della nostra macchina (per uno specifico lavoro)

in generale non c'è una macchina migliore di tutte le altre per tutti i possibili lavori ma in genere una macchina è migliore di un altra per una specifica applicazione anziché per un altra.
Per valutare le prestazioni di una CPU per una determinata applicazione si usa:

- tempo di risposta o di latenza che ci dice il tempo che serve per fare un determinato lavoro
- Throughput

che ci dice quanto lavoro riesce a fare il mio sistema nell'unità di tempo
Può succede che una CPU abbia un tempo di risposta elevato, ma anche un Throughput elevato segue che la nostra CPU riesce a fare nell'unita di tempo molto più lavoro di quella che riesce a fare un altra che ha un tempo di risposta inferiore, naturalmente la scelta di questi parametri dipende ancora una volta dall'ambito applicativo in cui dobbiamo andare a lavorare.
in Una Fotocamera digitale:

- tempo di risposta il tempo che passa da quando premiamo il pulsante a quando realmente l'immagine è salvata in memoria.
- Throughput

e il numero di foto che riesco a scattare in un secondo
I due parametri non sono direttamente collegati, magari una fotocamera con un tempo di risposta maggiore, riesce a scattare più foto in un secondo. (Quando si fa la panoramica)
In Altre parole in Throughput può essere elevato anche con un tempo di risposta grande (un altro esempio e quando eseguo tanti lavori in parallelo su più cpu ho un Throughput elevato molti lavori nell'unita di tempo, ma potrei avere un tempo di risposta della singola cpu molto grande).
Nelle valutazioni successive che andremo a fare non consideriamo il Throughput ma il tempo di risposta, in particolare quello che andremo a valutare sarà il tempo di CPU ovvero quanto tempo la CPU spende per eseguire un certo programma, il tempo di CPU non include il tempo speso per le operazioni di I/O, è soltanto il tempo di esecuzione sia in modo utente che in modo kernel del nostro programma.
Il tempo di risposta generico include anche il tempo speso dalla cpu per accedere hai dispositivi di I/O.
il tempo di CPU ci permette di valutare quale delle CPU è migliore per una data applicazione,

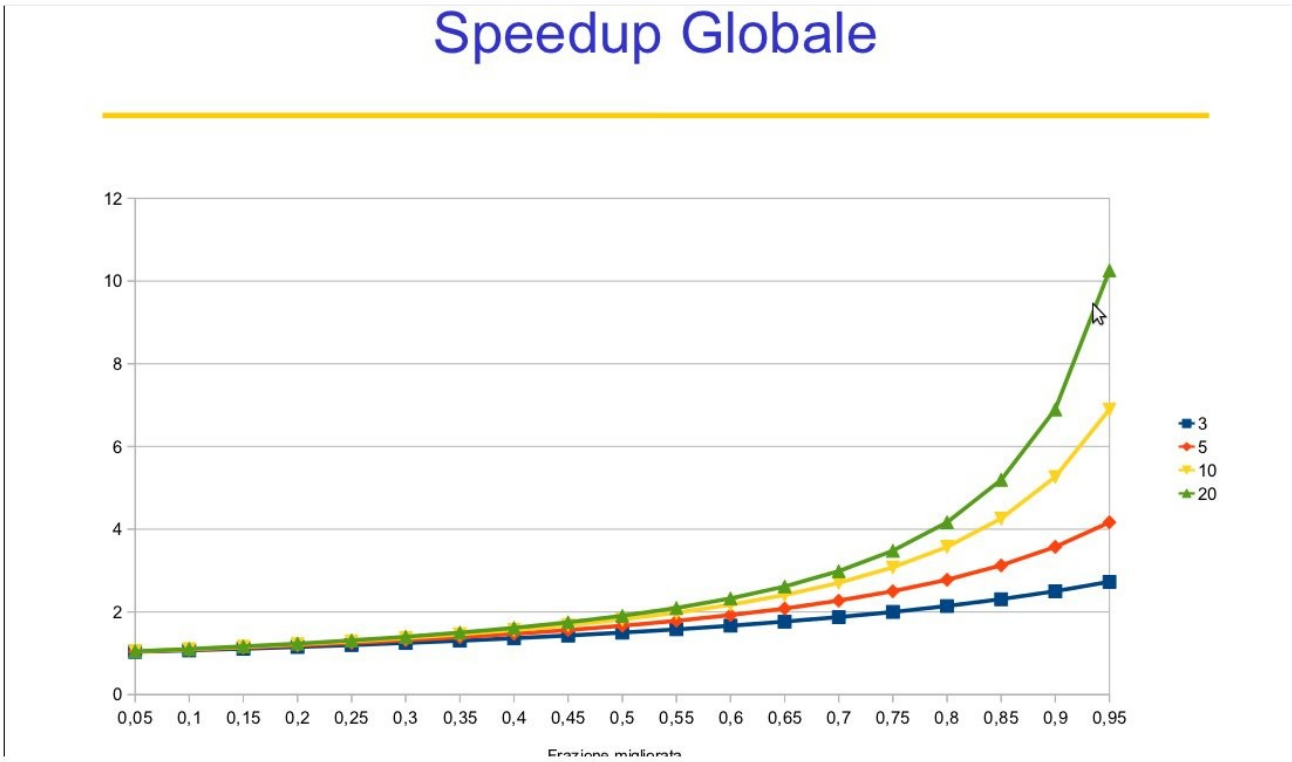
diverso dal tempo di risposta che serve per valutare l'efficienza di tutto il sistema nel suo complesso.
Dire che ma macchina X e n% più veloce della macchina Y significa:
(tempo di esecuzione di un dato programma sulla macchina Y)/(il tempo di esecuzione dello stesso programma sulla macchina X)= 1+n/100
tempo di esecuzione di Y=1/(prestazione di Y)
n=(prestazioni di X – prestazioni di Y)/(Prestazioni Y)
In genere il caso più frequente è anche il più semplice da rendere più veloce rispetto al caso meno frequente (istruzione di cpu).
Nella progettazione dobbiamo tenere contro di quella che è la legge di amdahl che dice che Il miglioramento di prestazione che può essere
ottenuto usando alcune modalità di esecuzione più
veloci è limitato dalla frazione di tempo nella quale
tali modalità possono venire impiegate
.
Quanto detto sopra significa il miglioramento di una parte del sistema ad esempio di 100 volte è limitato da quanto di frequente quella parte stessa del sistema viene utilizzata, è inutile migliorare di 100 volte una parte che viene usata magari solo nel 5% dei casi nel nostro sistema!!! perché il costo dell'hw migliorato magari sarà elevato senza avere noi un miglioramento significativo. Quindi noi se dobbiamo migliorare il nostro sistema dobbiamo migliorare la parte del sistema che viene usata più di frequente, questo miglioramento viene quantizzato dalla formula:
Speed up=(prestazioni quando miglioriamo una parte del sistema)/(prestazioni del sistema non migliorato)
ovvero
Speed up=(tempo di esecuzione senza miglioramento)/(Tempo di esecuzione con il miglioramento)
naturalmente questo rapporto deve essere maggiore di uno altrimenti non abbiamo avuto alcun miglioramento.
la frazione di tempo nel quale noi applichiamo il miglioramento in genere è minore di uno nel caso ideale è uguale ad uno, perché non tutte le istruzioni vengono migliorate.

$$\text{Tempo di esecuzione}_{\text{nuovo}} = (1 - \text{Frazione}_{\text{migliorato}}) \times \text{Tempo di esecuzione}_{\text{vecchio}} + \text{Frazione}_{\text{migliorato}} \times \text{Tempo di esecuzione}_{\text{vecchio}} / \text{Speedup}_{\text{migliorato}}$$

$$\text{Speedup}_{\text{globale}} = \frac{\text{Tempo di esecuzione}_{\text{nuovo}}}{\text{Tempo di esecuzione}_{\text{vecchio}}}$$

$$\text{Speedup}_{\text{globale}} = \frac{1}{(1 - \text{Frazione}_{\text{migliorato}}) + \frac{\text{Frazione}_{\text{migliorato}}}{\text{Speedup}_{\text{migliorato}}}}$$

Calcolatori Elettronici
Da quello che si vede nell'ultima formula si nota che lo speed up globale è fortemente condizionato dalla frazione di tempo in cui applichiamo il miglioramento
esempio di andamento dello speedup globale:



(Vedere Gli Esempi sugli esercizi sullo Speedup nelle slide il prof scrive alla lavagna)
lo speedup globale è limitato da 1/(1-frazione migliorato)

Lezione 11 Maggio 2011
Abbiamo visto che il tempo di CPU è il rapporto tra il numero di cicli della CPU e la frequenza di clock.
La frequenza ricordiamo che è data dall'inverso del periodo.
Aumentando la frequenza di clock diminuisce il tempo di CPU.

Per esprimere il tempo di CPU usiamo un parametro: il CPI.
Esso è il numero medio di cicli di clock per istruzione.
Esempio: se un programma ha 1000 istruzioni e impiega 3000 cicli di clock per terminarlo vuol dire che il CPI è 3000/1000=3, cioè mediamente sono impiegati tre cicli di clock per istruzione, ma in realtà non impiegano tutte le istruzioni tre cicli, esso è solo un valore medio.
Ci sono istruzioni piu complicate e altre meno che differiscono per il numero di cicli di clock impiegati.
Per esempio le moltiplicazioni impiegano piu cicli perche magari sono piu complesse e hanno piu latenza.

Il tempo di CPU è legato a ben 4 fattori:
N ->numero istruzioni eseguite.
CPI ->mediante valore medio, dipende pure dalla frequenza delle istruzioni eseguite.
T ->periodo clock.
f ->frequenza.

IMPORTANTE: le prime due dipendono dal programma e dal processore
le altre due dipendono solo dal processore.

Tcpu= N + CPI/f = N + CPI*T

N : dipende dal set istruzioni che il processore mette a disposizione e dunque varia da processore a processore. A parità di processore, dipende dal tipo di compilatore.
Può capitare che N sia basso ma CPI migliore dunque occorre ben scegliere quale processore usare.
Questo parametro da solo non è buono per scegliere un processore.
I processori RISC hanno un N inferiore.

f : dipende solo dal processore, cioè maggiore è la scala d'integrazione dei componenti e piu è maggiore
la frequenza.
Cioè migliorando la tecnologia diminuisce il ritardo cioe la latenza.
Dunque la latenza è dipendente dalla struttura interna del sistema.
La scelta dell'organizzazione interna è legata alla latenza,area,potenza , fattori spesso contrastanti tra
loro.
I processori RISC hanno frequenza maggiore della CISC, perche questi CISC implementano istruzioni
piu complesse che aumentano la latenza.

CPI : abbiamo detto che è il numero medio di cicli clock per istruzione.
Esso dipende dal programma e dal set istruzione del processore e dalla sua organizzazione.
Processori con poche istruzioni, hanno un piccolo CPI .

Per migliorare il CPI si usano due strategie:
__PIPELINE è un approccio a catena di montaggio, dove una istruzine è costituita da piu fasi.
Lo scopo è ottenere un CPI=1 ma nella realtà è >1

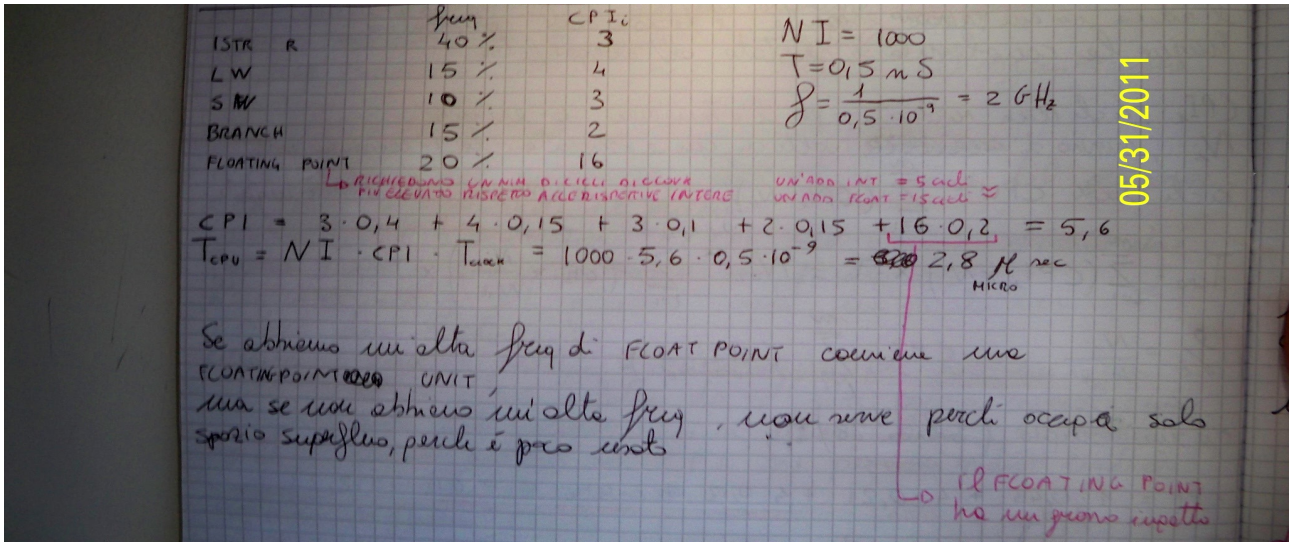
__PARALLELISMO di ISTRUZIONI è un approccio dove si eseguono piu istruzioni contemporaneamente, essa però è una cosa difficile da realizzare perchè tipicamente i programmi
non sono parallelizzabili e richiedono la sequenzialità. Anche se avessimo un processore con 4
unità che possono lavorare in contemporanea, al max se ne eseguono due istruzioni alla volta a causa
della dipendenza delle istruzioni.
Questo approccio puo essere SUPERSCALARE o VLIW.
-La SUPERSCALARE consiste che l'esecuzione non è in ordine. Ci sono istruzioni che iniziano in
un momento e finiscono in un altro e l'ordine di uscita è diverso da quello di ingresso.
Questo approccio aumenta la logica di controllo, l'area ma pure le prestazioni.
-La seconda VLIW ha un compilatore piu difficile ma processore piu semplice, è sempre in parallelo e l'esecuzione è ordinata e l'ordine è deciso dal compilatore.

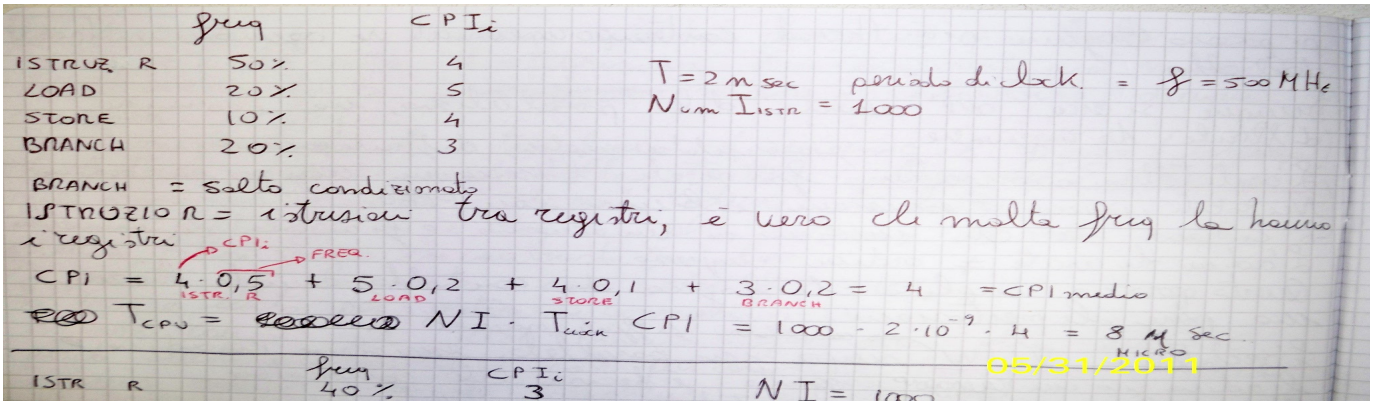
CPIi : cicli clock richiesti per l'istruzione i-esima.
Ni : numero di volte in cui è eseguita la i-esima istruzione.
cicli clock CPU = $\sum$ (CPIi * Ni) [i va da 1 a n] con n = numero istruzioni diverse eseguite.
Tcpu = $\sum$ (CPIi * Ni)*T = Nist * $\sum$(CPI * Ni/Nist)*T
fi=Ni/Nist .
CPI=$\sum$(CPIi * fi)
A slide 29 c'è un esercizio per misurare le prestazioni di due processori.
Si vede che per fare il confronto, il miglior strumento di confronto è Tcpu.

Occorre vedere quello. Potremmo avere una CPU1 che ha meno istruzioni della CPU2 ma se il Tcpu2 < Tcpu1 è sempre migliore il secondo.
A slide 30 per calcolare il Tcpu si usa il BANCH MARK. Esso è un insieme di programmi tipici nati per testare le prestazioni di un processore.

Un altro strumento di confronto è il MIPS= numero istruzioni eseguite/ tempo esecuzione*10^6= IC/Tcpu*10^6
non sempre un processore con MIPS piu alto di un altro vuol dire che è migliore.
Ci sono altre formule nelle slide successive.

NB: nonostante i vari strumenti, il dato piu importante è sempre e solo Tcpu.





Lezione 16 Maggio 2011
LEZIONE 11-05

L'ISA è un interfaccia tra Sw e Hw.I programmi vedono l'Hw tramite l'ISA.

Esso é fatto da tante parti (vedi slide) e sopra l'ISA ci sono il S.O e le applicazioni utente.

La scelta del set istruzione ha un impatto notevole sulle prestazioni del sistema.

Se il set è complesso si hanno problemi con ritardi e latenze, ma anche se è troppo semplice abbiamo problemi. Dunque serve una via di mezzo.

Il progettista sceglie in base a certi parametri:

-numero di operandi per istruzione.

-tipo,dimensione degli operandi.

-come fare i jump.

-altri fattori nelle slide(3).

L'ISA non ha nessun operando esplicito, vengono presi da una struttura stack tramite operazioni di PUSH.

Quando serve un operazione, vengono presi gli operandi dalla cima dello stack.

TOS =TOP of stack è puntatore alla cime dello stack e l'altro puntatore è un TOS+next.

L'ISA ha un accumulatore, che è un registro che viene usato implicitanmente.

L'ISA ha registri di vario tipo GENERAL PURPOSE e in genere sono di tre categorie:

-memoria_memoria le istruzioni e gli operandi sono mappati in memoria.Questa categoria risparmia in tempo ma il CPI aumenta troppo.

-registro_registro dove ogni operando è un registro.

-registro_memoria dove un operando è registro e uno è in memoria.

LEZIONE 16-05

memoria_memoria:

con 2 operandi si usano per risparmiare bit e uno dei due operandi è sia sorgente che destinazione.

con 3 operandi ci sono due sorgenti e una destinazione diversa.

registro_memoria:

con 2 operandi dove uno dei due è sia sorgente che destinazione

con 3 operandi dove ci sono due registri(uno destinazione e uno sorgente) e una memoria oppure ci puo essere due memoria e un registro.

registro_registro:

gli operandi sono solo registri ed è detto Load_Store perche per operare sui registri occorre caricare dalla memoria dei dati (load) e poi salvare con (store)

ci sono sempre due o tre operandi allo stesso modo del memoria_memoria.

Con 3 registri è meglio che 2.

Ci sono vari esempi di processori con architettura general purpose caratterizzati ognuno da:

-un certo numero di indirizzi di memoria che possono essere specificati nelle operazioni

-il num max di operandi

Esistono vari tipi di processori:

Ricordiamo che:

RISC hanno un ridotto set istruzioni; CISC hanno un elevato set istruzioni.

I processori piu recenti sono RISC con 0 indirizzi di memoria per le operazioni, gli indirizzi sono usati solo per load store e jump.

A=B+C può essere tradotta dal complitore usando 4 diversi approcci:

-STACK

è una colonna dove gli operandi sono impliciti cioe per fare la somma vengono usati i primi due elementi, si fa la somma tra loro e il risultato è messo

nuovamente in cima allo stack.

con PUSH si prende C e B e poi si fa ADD e il risultato è messo con POP in A

Abbiamo impiegato ben 4 istruzioni.

-REGISTRO_REGISTRO (Load Store).

se vogliamo far la stessa cosa serve copiare il contenuto di B in r1 e C in r2 e poi si fa AA e si mette la somma in r3 e con la store si va in A.

Abbiamo un numero di istruzioni sempre 4 ma in genere con codice piu complcato questo metodo aveva un numero di istruzioni minore rispetto al caso dello stack.

-MEMORIA_REGISTRO

dove il registro è un accumulatore.

-MEMORIA_MEMORIA

prevede una sola istruzione e come operandi ha tutte locazioni di memoria.

Guardando solo le istruzioni sembra che quest'ultimo approccio è migliore perchè ha meno istruzioni.

In realtà per valutare le prestazioni occorre notare anche altro:CPI ,periodo e frequenza.

Di solito i memoria_memoria hanno CPI piu elevato del registro_registro e periodo piu elevato pure.

Tipicamente chi usa questo l'approccio REGISTRO_REGISTRO ha un set istruzioni piccolo ed essendo poche le istruzioni la decodifica è veloce, la latenza è piccola e alla fine CPI piccolo, periodo piccolo e quindi anche se le istruzioni sono di piu nel registro_registro alla fine è meglio questo che memoria_memoria.

Si preferiscono processori con piccolo set istruzioni xke è meglio apportare le migliorie.

Nel corso del tempo si vede che i processori si evolvono tutti verso la stessa direzione LOAD-STORE cioe REGISTRO_REGISTRO.

I vantaggi delle architetture registro registro rispetto le altre :

-non dobbiamo accedere alla memoria e quindi operazioni piu veloci.

-architettura di tipo stack ha un ordine in registro registro invece non occorre ordine per avere risultato corretto.

-densita del codice aumenta, codice piu compatto e meno memoria. Occorre meno bit per fare le istruzioni.

Il progettista affronta delle problematiche per decidere come fare l'istruciton set:

-L'indirizzamento della memoria è il piu grosso e importante.

Esso si ha in ogni tipo di gestione che sia memoria o registro.

L'indirizzo è espresso in byte, e quel byte è proprio il primo byte da leggere o scrivere e dopo il primo ci sono gli altri che saranno usati.

Il fatto che l'indirizzo è espresso in byte crea problemi perchè se vediamo la memoria come tante righe fatte da 4 byte, un primo problema è capire

all'interno della griglia come i byte sono ordinati.

Gli approcci usati per gestire questo problema sono LITTLE ENDIAN o BIG ENDIAN.

Entrambi sono usati e occorre conoscere l'ordinamento dei byte all'interno delle word se no si fanno errori.

Se si usa uno o l'altro la traduzione è diversa.

Un altro problema è che gli indirizzi generati dalla cpu che sono espresi byte, se sono ALLINEATI o NON ALLINEATI.

-un indirizzo è ALLINEATO se l'indirizzo prodotto dalla cpu è multiplo del numero di byte di ogni word.(se una word è fatta da 4byte un indirzzo è allineato se è multiplo di 4) occorre che gli ultimi due bit siano zero cosi sono indirizzi multipli di 4.

nell'allineato gli indirizzi sono quelli come 0.4.8.12.16.20 ecc per word di 4 byte.

se gli indirizzi sono NON ALLINEATI, il primo byte non è un multiplo di 4 ma uno qualsiasi.

es se l'indirizzo è 1 è non allineato.

questo sempre perche consideriamo word di 4 byte.

Questo approccio non lineare è piu flessibile,perchè non ci serve piu un indirizzo come multiplo, però ovviamente perdiamo in termini di complicazione,cioe servono piu acccessi.

Se noi abbiamo un allineamento di indirizzi cioe sempre multipli di 4 8 ecc , la decodifica è molto semplice, si fa un accesso unico alla memoria si prende il dato e si utilizza, invece nel non allineato si devono fare piu cose ed è complicato.

le modalita di indirizzamento sono varie:

modalita REGISTRO dove tutti operandi sono registri è la modalita che è piu rapida,e permette di decodificare istruzioni con poche bit.

modalità IMMEDIATO quando si inizializza un registro(quando si da un valore iniziale a un registro: add R3,R0,5 dove R0 è un registro che da sempre valore 0,questa istruzione assegna al registro R3 il valore 5 perche 5+0=5) o qnd si lavora con delle costante (somma tra registro 3 e 20).

modalità DISPLACHEMENT usata in assembly, un accesso in memoria il cui indirizzo è somma tra registro e qualcos'altro.

modalità SCALABILITY dove si somma a un registro una locazione di memoria il cui valore era stato conservato ad un altro registro,è simile al displacement.

modalità INDICIZZATO dove la locazione di memoria è la somma del contenuto di due registri.

modalità INDIRIZZAMENTO DIRETTO o ASSOLUTA, mettiamo l'indirizzo preciso della locazione di memoria che vogliamo accedere,è simile all'IMMEDIATO ma quello serve pochi bit in questo servono piu bit se un indirizzo è grande.

modalità meno usate MEMORY DEFERRED dove un registro contiente puntatore a puntatore.Necessita di tante operazioni per ottenere cio che ci serve

modalità AUTOINCREMENTO o DECREMENTO, dove oltre a un accesso in memoria si fa un incremento del valore di un registro

ci sono altre modalità poco importanti e poco usate.

Tutte queste modalità sono TIPO ma nei processori piu recenti ne sono usate solo una piccola parte.

Quando si sceglie una modalita si fanno delle misure e attraverso esse si capisce quali sono utili o meno.

Nel caso di 5 modalità usando 3 applicazioni si vede come in ascissa c'è la frequenza con cui

vengono usate queste modalita al variare delle applicazioni, tra le diverse applciazioni cambia molto,per esempio DISPLACEMENT ha diversa frequenza in base alle applicazioni,stessa cosa vale anche per le altre modalità.

Al di la delle applicazioni si vede che ci sono tendenze.

Le IMMEDIATE e DISPLACEMENT sono le piu frequentemente usate dalle applicazioni. In media ricoprono circa il 75% dei casi.

Le altre modalità hanno uso piu basso,percio se uno vuol progettare set istruzioni con costo ragionevole è meglio implementare le modalità piu usate, soprattutto quueste due su citate e la SCALABILTY che assieme coprono il 90% dei casi.

Uando queste 3 si riduce la latenza e si rende il processore piu veloce e cpi piu basso.

Queste modalità sono le uniche implementate nei moderni processori.

Non vuol dire che quello che si fa con certe modalita non si possa fare con le altre,è solo che con le altre si fanno piu istruzioni,ma nel totale le 3 bastano.

Nelle DISPLACEMNT c'è un valore accostato, questo valore è rappresentato con un certo numero di bit. Se il valore è alto si usano molti bit se invece è basso se ne usano pochi.

Lezione 18 e 23 maggio

Abbiamo visto che per quanto riguarda il numero di bit che possono servire per codificare l'immediato di una istruzione, circa 80% delle volte si usano 16 bit per rappresentare l'immediato, in caso in cui, circa il 20% delle volte, non riusciamo a rappresentare l'immediato, noi possiamo sostituire l'istruzione con un immediato con un'istruzione con i registri, ma cosi usiamo più istruzioni. Visto che si tratta del 20% è preferibile spendere qualche istruzione in più al posto di appesantire il programma.
Se, invece andiamo a usare un numero di bit maggiore, questo vale a dire utilizzare, nella codifica dell'istruzione, uno con base maggiore; nel caso in cui il parallelismo del nostro calcolatore riesce a supportare una codifica con una base maggiore, quest'ultima non provoca effetti sulla latenza sull'esecuzione e questo è un vantaggio, ma se il nostro calcolatore non potrebbe supportare una codifica maggiore questo bisogna realizzare una fare di fetch con più accessi in memoria (mentre prima facevamo un unico accesso), ma in questo modo leggiamo più volte che complica le prestazioni (introduciamo ritardi nell'esecuzione).

Vediamo come la maggior parte delle istruzioni eseguite (come l'instaction set del processore X87)

° Rank	instruction	Integer Average Percent total executed
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	Total	96%

Come possiamo notare dalla slide gia con poche istruzioni riusciamo ad arrivare al 90% di frequenza di esecuzione del programma, dunque con poche istruzioni riusciamo a coprire gran parte del codice che serve per eseguire un programma. In questo modo riduciamo la latenza e riusciamo ad ottimizzare il codice
All'interno di un qualsiasi programma abbiamo istruzioni di salto che alterano l'esecuzione sequenziale.
All'interno di un programma abbiamo diversi tipi di salti, salti condizionati, oppure chiamate di sottoprogrammi
Alcuni esempi sono:

```
JMP    DEST      ; Diretto o relativo a PC
JZ  wait         ; Di solito relativo a PC
callsub          ; Di solito diretto
BR R30           ; EA destinazione = R30
```

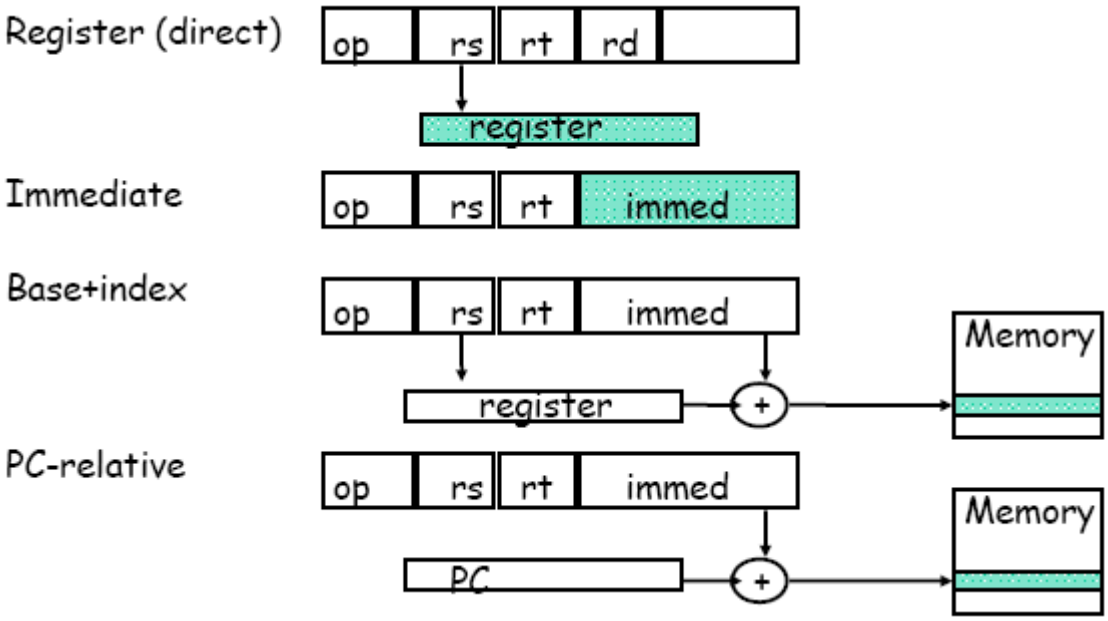
Insieme all'istruzione di salto.. ci sono le istruzioni di ritorno: ogni istraction set avrà una specifica istruzione
Abbiamo 2 casi

Indirizzamento diretto (solitamente in 32 bit), assoluto o relativo in cui dove l'indirizzo di salto è un indirizzo assurdo che lo andiamo a specificare con un immediato…
La soluzione alternativa è quella che viene chiamata Branch Dispacement in cui il salto viene fatto in avanti o all'indietro rispetto alla posizione corrente del PC (Program Counter).
Il salto Condizionato è molto usato, soprattutto per codificare i cicli (while, do while, for).
Un ciclo è formato solitamente da 10-15istruzioni ✳ questo vuol dire che il numero di bit usati per codificare il salto è molto limitato (7-8 bit).. naturalmente aumentato il numero di istruzioni si devono usare + bit

Nel caso che abbiamo un bisogno di usare + di 16 bit ✳ usiamo + istruzione per descrivere il salto

- Modalità di indirizzamento dei dati più importanti:
  - Displacement, Immediate, Register Indirect
- Dimensione dello spiazzamento:
  - dai 12 ai 16 bit
- Dimensione degli operandi immediati:
  - dagli 8 ai 16 bit

Nell'MIPS tutte le istruzioni hanno lo stesso numero di bit, cioè 32 bit



Per questo processore abbiamo un capo codice operativo che ha una dimensione fissa che solitamente è di 6 bit.
Abbiamo per istruzione di tipo registro diretto che sono denominati con rs, rt, rd che occupano 5 bit ciascuno, dunque utilizziamo 21(5+5+5+6) bit e i bit rimanenti spesso non vengono utilizzati(dunque abbiamo uno spreco)

Nel caso di istruzioni immediato non abbiamo il registro rd ma introduciamo "immed" che misura 16 bit.

Nel caso Base+Index che il valore del registro viene sommato con il'immed e allocato in memoria

Nel caso PC-relative (quello meno raro e più complesso da implementare), si ha un aggiornamento del PC.

Un altro caso il PowerPC che ha una struttura simile dell'MIPS: lavora con 32 bit. E possiede un codice operativo.
L'unica differenza è quella legata all'OE e RB che servono per amplificare il OverFlow.
Abbiamo un campo ALU che ci suggerisce quale operazione si svolgerà.
Nel caso dell'immediato la codifica la codifica è la stessa di quello di MIPS.
Il salto condizionato viene dato da Li che utilizza 32-8 = 24 bit

La struttura Intel di cui l'istruzione può essere preceduta da flussi..
Nei diversi capi ci sono 0 e 1 che indicano il numero di byte per ciascuno campo dell'istruzione..
Inoltre abbiamo una maggiore flessibilità

The DXL ISA

Questo processore è caratterizzato dal fatto che possiede un istruction set di tipo Load/Store.
Le istruzioni sono di tutte della stessa lunghezza, dunque abbiamo una codifica più semplice.
Alcune caratteristiche principali di questo processore sono dal fatto che abbiamo 32 registri General Pur pose e 32 registri Floating Point (dunque con dobbia processione).
La lunghezza di una word è di 32 bit.
Vediamo come sono organizzati i registri:
Il primo registro, R0 , dei 32 registri General Pur pose da 32 bit , contiene il valore 0 che non può essere modificato, mentre l'ultimo registro, R31, serve per conservare l'indirizzo di ritorno per le istruzioni JAL e JALR, in fine abbiamo gli altri 30 (R1-R31) che sono dei reali GP register
La JAL definisce un immediato, mentre la JALR l'indirizzo è definisce un registro.
Sono registri a 4 byte
L'ordinamento dei byte è fatto

È possibile accedere o a un singolo byte(cioè al primo), o a 2 byte (in questo caso si ha un Halfword) o a tutti e 4 byte(si ha un fullword).

Ci sono tre registri speciali:

- ***PC***, Program Counter, contiene l'indirizzo dell'istruzione dal leggere dalla memoria (32 bit)
- ***IAR***, Interrupt Address Register, mantiene l'indirizzo di ritorno di 32 bit del programma interrottoquando una istruzione *TRAP* viene eseguita (32 bit)
- ***FPSR***, Floating-Point Status Register, utilizzato nei conditional branch per valutare il risultato di una operazione FP (1 bit)

Il DXL contiene 32 istruzioni divere in 6 classi:

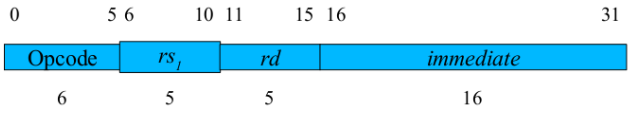
1. load & store instructions
2. move instructions
3. arithmetic and logical instructions
4. floating-point instructions
5. jump & branch instructions
6. special instructions

- es. le istruzioni che lavorano con i registri principalmente per l'accesso in memoria
- servono per spostare i valori di un registro all'altro
- dobbiamo intendere operazione di somma, prodotto, OR, And (Sono di tipoR e di tipo I)
- quando dobbiamo fare operazioni di virgola mobile
- istruzioni di salto condizionato
- istruzioni speciali che non vediamo

Il formato delle istruzioni si possono distinguere in 3 Categorie

Tipo-I : che presentano un campo immediato
Tipo-R: lavorano con i registri e non possiedono un immediato
Tipo-J: servo per codificare le istruzioni di salto

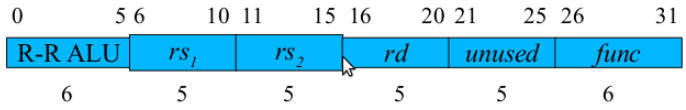
Le istruzioni di Tipo I (I=immediate) è utilizzata per la codifica delle istruzioni come le Load/store, JR, Jalr, ALU con immediato.



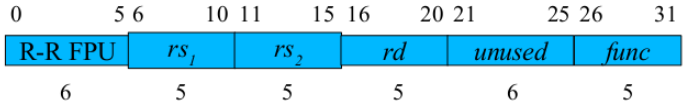
- ***Opcode***: istruzione DLX da eseguire
- ***rs*$_1$**: sorgente per l'ALU, indirizzo base per le Load/Store, registro da testare per i conditional branches, indirizzo destinazione per JR & JALR
- ***rd***: destinazione per la Load e le operazioni ALU operations, sorgente per la store. (Non usato per conditional branches, JR e JALR)
- ***immediate***: spiazzamento per calcolare l'indirizzo delle load e delle store, operando per l'ALU, spiazzamento esteso in segno da sommare al PC per calcolare l'indirizzo destinazione di un conditional branch. (non usato per JR e JALR)

Le Istruzioni di tipo R (R=Register) sono utilizzate per le operazioni riguardanti l'ALU e per scrive o leggere sui registri speciali.
La func ci dice l'operazione che dobbiamo fare (somma, prodotto, ecc..)
Gli indici di rs1 e rs2 non sono indici a intero ma a floating-point

Le istruzioni di tipo J (J=jump)
6 bit sono utilizzati per il codice operativo gli altri 26 bit per lo spiazzamento con segno
In questo caso non c'è un intirizzimento assoluto




```
add  r1,r2,r3 ;rd=r1, rs1=r2, rs2=r3
```




```
addf f1,f2,f3 ;rd=f1, rs1=f2, rs2=f3
```

PIPELINE

Supponiamo di lavorare con dei capi d'abbigliamento riguardo alla pulizia
PASSO 1: dobbiamo lavare l'abito con la lavatrice
PASSO 2: Se non siamo in Sicilia utilizziamo una asciugatrice per asciugare l'abito
PASSO 3: stiriamo l'abito
Ognuno di queste 3 attività coinvolge 3 risorse diverse: LAVATRICE, ASCIUGATRICE, FERRO DA STIRO.
Supponiamo che il lavaggio occorrono 30min, per asciugare 40min e per stirare 20min
Se operiamo in un modo sequenziale (guarda animazione) e supponiamo di lavorare con 4 capi:
1 abito ✳ (laviamo asciughiamo ecc..) passano 90 minuti
2 abito ✳ (laviamo asciughiamo ecc..) passano 90 minuti
3 abito ✳ (laviamo asciughiamo ecc..) passano 90 minuti
4 abito ✳ (laviamo asciughiamo ecc..) passano 90 minuti
TOTALE TEMPO TRASCORSO 6 ORE!!!

Questo tipo si soluzione non è efficiente perchè durante l'esecuzione di una delle attività le altre rimangono ferme.
Se utilizziamo una esecuzione pipeline, vale a dire utilizzare la risorsa appena è disponibile, (guarda animazione) vediamo che abbiamo una ottimizzazione del tempo, al posto di utilizzare 6 ore abbiamo utilizzato 3 ore e 10 minuti ✳ abbiamo migliorato di circa del 50%.

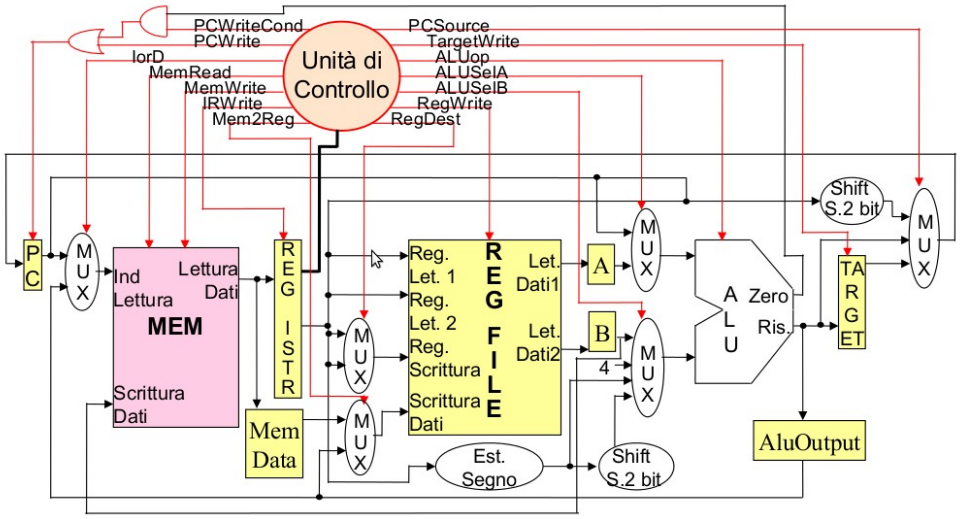
Il tempo di esecuzione della singola istruzione non diminuisce (alla fine per ogni abito sempre 1 ora e 10 impieghiamo), ma il tempo medio di esecuzione delle istruzioni si riduce in un fattore N, questo è un caso ideale.
Inoltre il throughput migliora di un fattore N questo perché, ed è la differenza sostanziale, il CPI del pipeline è = 1, mentre in quello sequenziale è uguale a N.
Inoltre il pipeline non riduce la latenza del singolo task ed, caratterista fondamentale più task operano insieme.

$$CPUtime_{Pipe} = IC \times CPI_{Pipe} \times T_{CK} = IC \times 1 \times T_{CK}$$
$$CPUtime_{Seq} = IC \times N \times T_{CK} = N \times CPUtime_{Pipe}$$

$$Speedup = \frac{CPU_{TIME\ Sequenziale}}{CPU_{TIME\ Pipeline}} = N$$

Se tutte le N fasi hanno uguali, questo è un caso ideale, in quanto lo speedpup è = N
Ma se le N fasi hanno tempi diversi lo speedpup non è più N ma è un valore più basso.
Se aumentare il numero di fasi ✳ aumentiamo le istruzioni ✳ aumentiamo la complessità della gestione pipeline ✳ più difficilmente ci avviciniamo al caso ideale.
Ricapitolando:
Ipotizzando di dividere le fasi di esecuzione dell'istruzione in N fasi e ipotizzando che le fasi sia correttamente bilanciate, lo speedup ottenuto considerando una versione sequenziale, risulta pari ad N, questo è un caso ideale.

Andiamo a guardare quali sono le codifiche che dobbiamo apportare a nostro processore sequenziale per farlo diventare un processore pipeline..



Il processore sequenziale: La memoria è utilizzata sia per fare lo fetch dell'istruzione sia per leggere o scrivere dei dati.
Il MUX può ricevere due indirizzo, o provenienti dal PC o dall'ALUOutput.
L'ALU viene utilizzata: per l'aggiornamento del PC, è utilizzata per la fase di decode per calcolare in anticipo l'indirizzo target, è utilizzata durante la fase di esecuzione e per calcolare l'indirizzo della locazione in memoria.
In caso di branch quest'ALU verrebbe utilizzata per fare il calcolo della condizione.
In questo schema la memoria e l'ALU vengono utilizzate per + fasi
+ fasi non possono lavorare sulla stessa risorsa.. e le soluzioni sono 2:

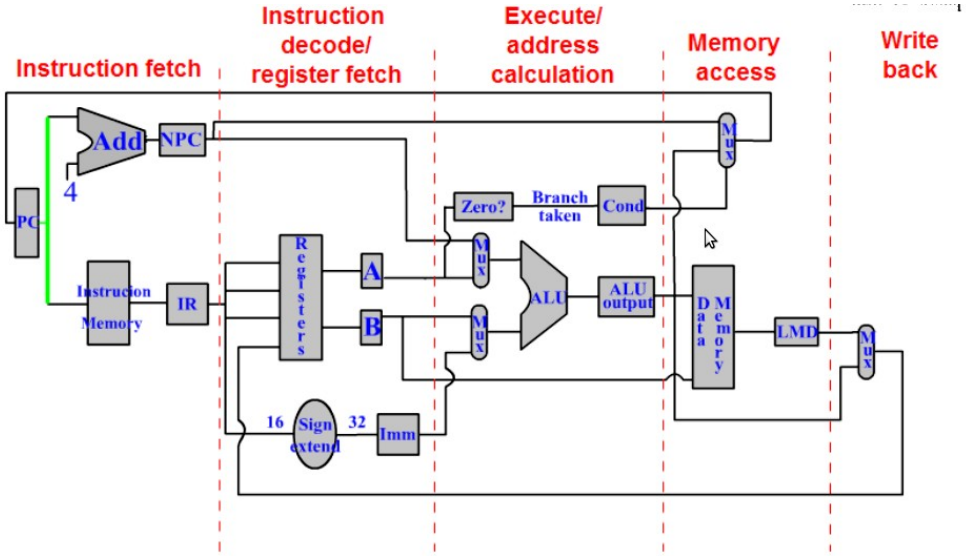
- sequenziale l'accesso a questa risorsa✳ questo vuol dire ridurre le prestazioni del sistema
- duplicare le risorse ✳ i due attori possono agire contemporaneamente

Ricorda: 1 risorsa ✳ è possibile fare 1 operazione a volta, dunque se voglio che due attività

agiscono sulla stessa risorse, bisogna che uno delle due deve aspettare

Se un immediato deve essere eseguito in due cicli di clock consecutivi ✳ non è possibile eseguire una struttura pipeline, tranne se andiamo a duplicare l'immediato o il registro, o meglio alla fine della fase di decodifica andiamo a copiare i dati in altri registri, cosi durante la fase di esecuzione abbiamo i dati..
In questo modo possiamo fare contemporaneamente la decodifica e l'esecuzione.
Se guardiamo il PC, in questo schema , il Pc viene mandato da questo Mux e viene utilizzato durante la fase di fetch per aggiornarsi se stesso, ma potrebbe servire per calcolare l'indirizzo target. Dunque il PC serve sia per la fase Fetch sia per la fase di Decode ✳ questa cosa non va bene perche dovrebbe essere mantenuto in due cicli di clock consecutivi ✳ e' come la fase di fetch e di decode siano un'unica cose.
Per risolvere il problema duplichiamo il PC:
Dopo la fase di fetch , andiamo a memorizzare il valore del PC, a questo punto durante la fase di decodifica possiamo fare l'aggiornamento del PC per un'altra istruzione, il pipatane in questo caso richiedere l'aumento delle risorse disponibili. Dove viene memorizzato il PC? Nel NPC (c'è scritto nella slide).

Abbiamo un blocco(fase) che va la catturare la condizione, viene preso il contenuto del registro A e vedere quanto vale, della fase di esecuzione abbiamo il calcolo dell'esecuzione e nella fase dell'accesso della memoria abbiamo la memoria dove viene scritto il risultato.



Abbiamo 4 blocchi, cioè 4 fasi.
Tra una fase all'altra risono dei registri, questo per rendere possibile il pipeline (alcuni per fare la duplicazione). Il primo registro IF/ID possiede in ingresso due linee che corrispondono al registro NPC e l'altro ingresso IR.
In questo schema ALU viene duplicata, questo per utilizzarla in 2 fasi diverse.
A valle del sommatore abbiamo un MUX.
Nella fase di decodifica l'uscita viene distribuita in diverse parti IR1..15, IR6.10 che corrispondono alle due registri sorgenti dell'istruzione.
In seguito sono riportate le varie istruzioni e comandi

Instruction Fetch (IF)

IF/ID.IR <-- Mem[PC]

All' interno del pipeline-register **IF/ID** c'è il campo **IR** ed il campo **NEWPC (NPC)**
Nel campo IR viene memorizzato il valre di PC contenuto in memoria.

IF/ID.IR <-- Mem[PC]

IF/ID.NPC,PC<--if(EX/MEM.cond (EX/MEM. ALUOutput) else (PC+4)

All' interno del pipeline-register **IF/ID** c'è il campo **IR** ed il campo **NEWPC (NPC)**
Nel campo IR viene memorizzato il valre di PC contenuto in memoria.
I campi PC ed NPC vengono aggiornati. Se il valore della condizione nel registro EX/MEM è vera si assegna a PC e NPC il campo ALUOutput di EX/MEM che sarebbe l' indirizzo target ; oppure metto PC = 4.

Instruction Decoder (ID)

ID/EX.A <-- Regs[IF/ID.IR$_{6..10}$]

ID/EX.B <--

I registri indirizzati da **IF/ID.IR$_{6..10}$** e **IF/ID.IR$_{11..15}$** vengono letti e memorizzati rispettivamente in **ID/EX.A** e **ID/EX.B.**
Contemporaneamente viene eseguita l'estensione in segno da 16 a 32 bits del dato memorizzato in **ID/EX.Imm** e scritto nel registro di pipeline **ID/EX.Imm**

Branch Instruction

EX/MEM.ALUOutput <-- ID/EX.NPC + ID/EX.Imm

Il contenuto del registro **ID/EX.NPC** è sommato col contenuto del registro **ID/EX.Imm** e il risultato è scritto in **EX/MEM.ALUOutput**.

EX/MEM.cond <-- (ID/EX.A *op* 0)

Contemporaneamente viene confrontato il contenuto del registro **ID/EX.A** con **0** e il risultato è posto in **EX/MEM.Cond**

Memory access (MEM)

MEM/WB.IR <-- EX/MEM.IR

Il valore contenuto nel registro **EX/MEM.IR** si propaga fino al registro **MEM/WB.IR**
Quindi, a seconda che il tipo di operazione sia una *"ALU Instruction"* o *"Load e Store Intruction"* ci sono due possibili sviluppi.

Memory access (MEM)

ALU Instruction

MEM/WB.IR <-- EX/MEM.IR

MEM/WB.ALUOutput <-- EX/MEM.ALUOutput

In caso di *ALU Instruction* il contenuto del registro contenuto in **EX/MEM.ALUOutput** si propaga fino al registro **MEM/WB.ALUOutput**

ALU Instruction

Regs[MEM/WB.IR$_{16..20}$] <--- MEM/WB.ALUOutput

Il contenuto del registro **MEM/WB.ALUOutput** viene letto e memorizzato nei registri puntati dal contenuto in **MEM/WB.IR$_{16..20}$.**

or

oppure:

Regs[MEM/WB.IR$_{11..15}$] <--- MEM/WB.ALUOutput

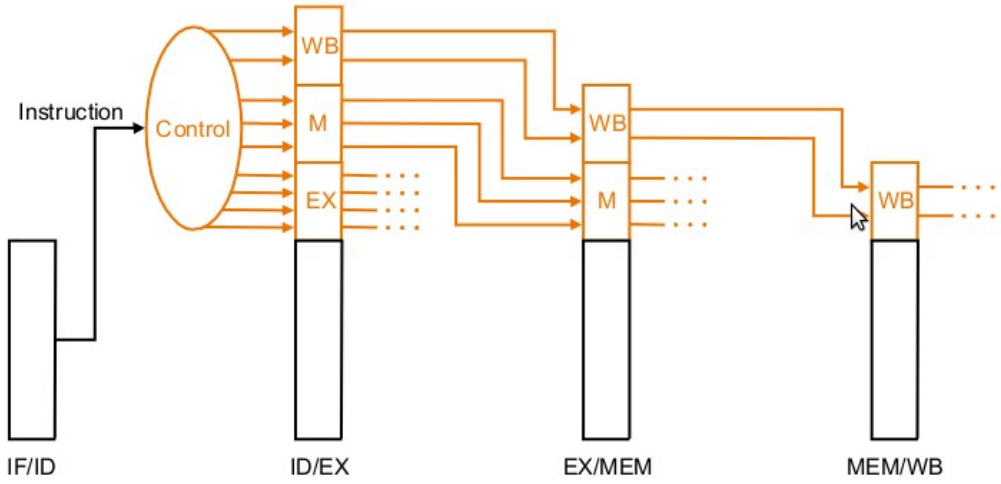
Il contenuto del registro **MEM/WB.ALUOutput** viene letto e memorizzato nei registri puntati dal contenuto in **MEM/WB.IR$_{11..15}$.**

Load/Store Instruction

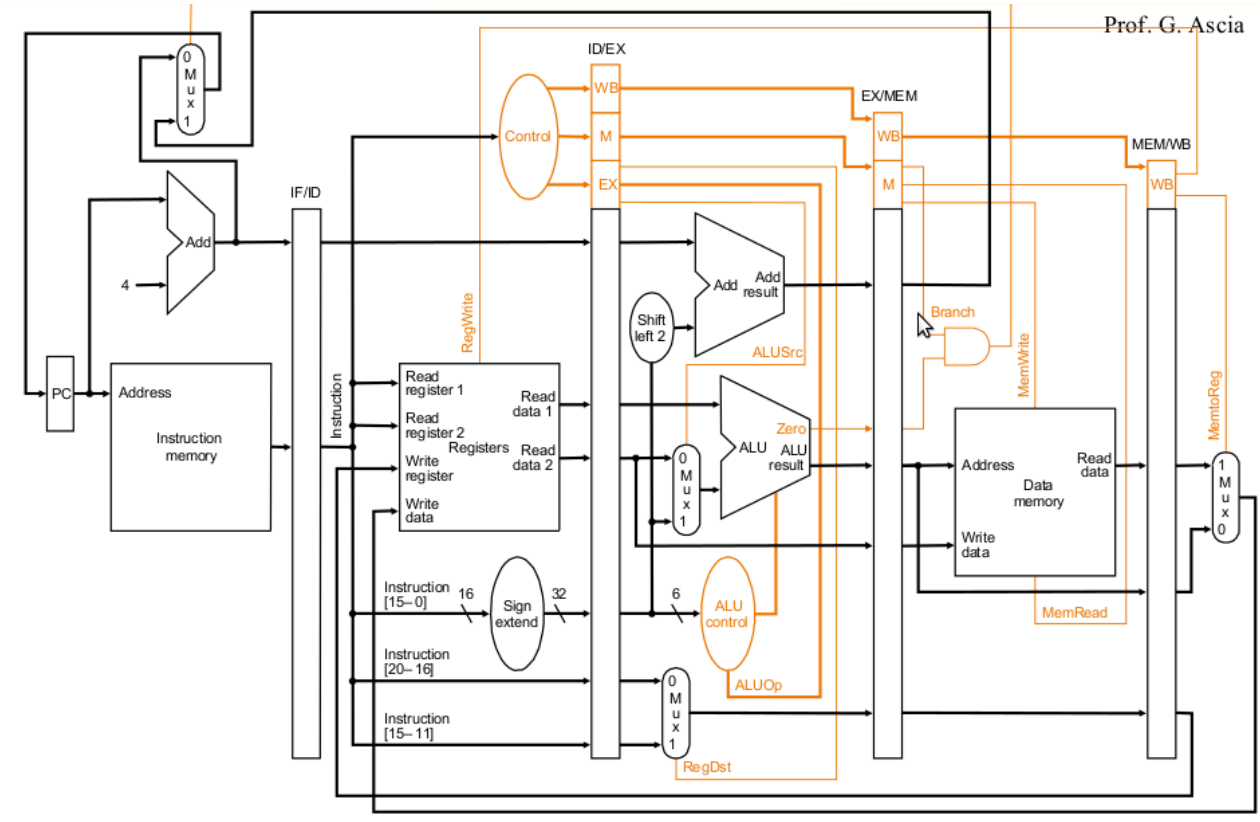
Regs[MEM/WB.IR$_{11..15}$] <-- MEM/WB.LMD

Il contenuto del registro **MEM/WB.LMD** viene letto e memorizzato nei registri puntati dal contenuto in **MEM/WB.IR$_{11..15}$.**

UNITA' DI CONTROLLO IN CASO PIPELINE



Al termine del fetch, noi mandiamo l'istruzioni che deve essere decodificata, ma prima andrà all'unità di controllo, che andrà a generare tutti i segnali di controllo che servono al datapath in tutte le fasi successive. Alcuni di questi segnali verranno utilizzati subito, gli altri verranno ricopiati negli registri successivi



Questo è uno schema più completo di processore pipeline dove abbiamo messo il datapath e l'unità di controllo.
Abbiamo Il PC, il MUX, il Sommattore.
Durante la decodifica vengono prodotti vari segnali, tra cui il segnale di controllo per il branch (memorizzato nel registro M) che nella fase ID/EX non serve, dunque verrà copiato nel registro della fase successiva, vale a dire EX/MEM dove verrà utilizzato.
Nella fase di esecuzione abbiamo l'ALU, il sommatore che calcola l'indirizzo target.
Ogni registro che si contiene nella fase di esecuzione riposta li stessi valori dei registri della fase di ID/EX che a loro volta solo stati creati dall'unità di controllo; I MUX che troviamo ci rilevano gli registi di destinazione (con l'aiuto del controllore).
L'altro Controllore (ALU Control) ci rileva quali dei 2 possibili ingressi dobbiamo mandare all'ALU che il suo risultato viene mandato nel registro EX.
Riguardo l'accesso della memoria abbiamo (ovviamente) la memoria, i segnali di controllo MemRead e MemWrite, che ci indicano se dobbiamo fare un accesso alla memoria in lettura o in scrittura. Questi segnali di controllo insieme al Branch si trovano nel campo M, che erano stati prodotti dalla fase di decodifica.
Nella fase W/B abbiamo due possibili valori da scrivere: o il dato proveniente dalla memoria o dall'ALU.

Separando memoria dati -memoria istruzioni non abbiamo conflitti, ma nel caso che usiamo un'unica memoria, le prestazioni che dobbiamo avere per un processore pipeline devono essere maggiori, es. nella fase di fetch deve essere eseguita in ogni ciclo di clock, mentre in quella sequenziale la fetch deve essere eseguita ogni 5 cicli.

Riguardo alla memoria dobbiamo tener conto di questi due parametri: Tempo accesso e tempo di ciclo.
Il tempo di accesso ci dice quanto tempo passa da quanto mandiamo in esecuzione il PC al risultato.
Il tempo di ciclo: Quanto tempo passa tra un accessoal successivo

Nel caso sequenziale il tempo di ciclo è di 5 cicli di clock
Nel caso pipeline il tempo di ciclo deve essere 1 ciclo di clock

Il Pc nella mansione pipeline deve essere aggiornato (nel fetch) ad ogni ciclo di clock, ma quanto andiamo ad eseguire un istruzione di branch potrebbe accadere, che l'aggiornamento decorre contemporaneamente (visto che quando facciamo lo branch aggiorniamo il PC) ✳ dunque abbiamo un conflitto tra 2 istruzioni: 1 che si trova nella fase di fectc che vorrebbe aggiornare il PC, mentre il branch che vuole pure lui aggiornare il PC. Dunque dobbiamo gestire questo litigio. Ma lo vedremo + avanti.

La presenza dei Hazzard (cioè i conflitti) ci fa allontanare dal caso ideale.

•I conflitti (Hazard) impediscono che una istruzione venga eseguita nel ciclo di clock atteso

-Structural hazards: Le risorse HW non supportano alcune combinazioni di istruzioni

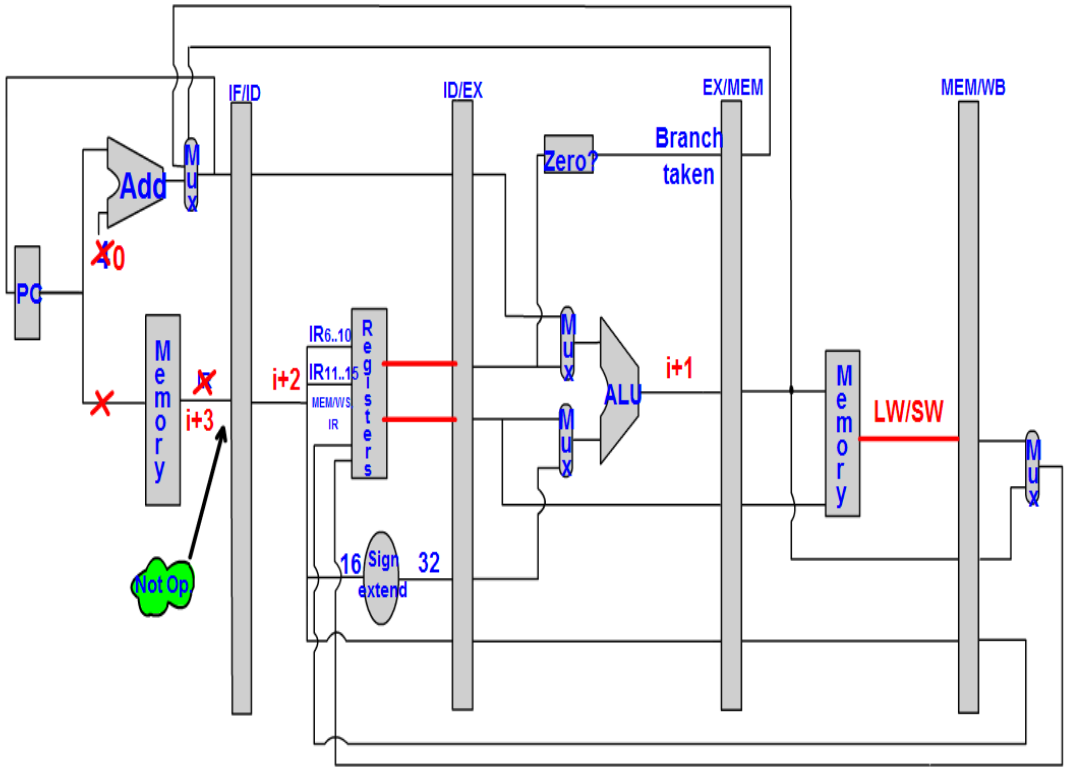
-Data hazards: Un'istruzione dipende dal risultato di una istruzione che è ancora nella pipeline

-Control hazards: Pipelining di branch e altre istruzioni che cambiano il PC

La soluzione più semplice è quello dei cicli di clock di stallo
Il ciclo di clok di stallo blocca l'esecuzione ✳ + cicli di stallo facciamo + peggioriamo le prestazione ✳ CPI > 1

Lezione 25 Maggio 2011

Schema del processore pipeline in cui si vede come andare andare a implementare lo stallo nel caso di hazard strutturale.



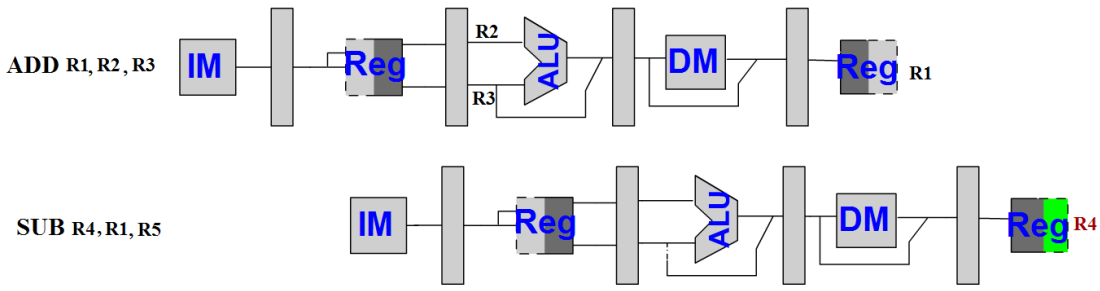
Nella fase di accesso alla memoria c'è un'istruzione che potrebbe essere una load o una word, nella fase di fetch c'è un'istruzione che dovrebbe accedere alla memoria (nel disegno, la memoria è rappresentata con due blocchi, ma bisogna immaginarla come se fosse un blocco unico). Alla memoria non arriva il contenuto del PC, ma arriverà l'indirizzo che è stato calcolato dall'ALU. A questo punto, perchè possa essere introdotto un ciclo di clock di stallo nella fase di fetch, bisogna

evitare che ciò che viene dalla memoria venga scritto nel registro IF/ID, cosa che si ottiene memorizzando in questo registro una configurazione di bit che corrisponde all'istruzione not op (=not operation), che avrà come effetto quello di non alterare lo stato del processore, ossia non scriverà nei registri o nella memoria. Contemporaneamente, per evitare che l'istruzione che dovrebbe essere eseguita nella fase di fetch venga persa bisogna inibire l'incremento del PC. In figura si vedere come il valore 4 all'ingresso dell'add sia tagliato e vi sia uno 0, proprio per evidenziare che non vi è incremento del PC. Questo può accadere se al PC non arrivano segnali di abilitazione alla scrittura e quindi facendo in modo che venga inibita la scrittura. In alternativa dovremmo far arrivare al mux (quello che c'è all'uscita dell'add in uscita al PC)add+0, ma questo vorrebbe dire che nel secondo ingresso secondo ingresso ci sia 4 o 0, quindi ci vorrebbe un mux comandato da un segnale di controllo che potrebbe essere utilizzato per abitilitare la scrittura del PC. Nel ciclo di clock successivo, l'struzione di load passerà in writeback, il nop passerà alla fase di esecuzione e l'istruzione nella fase di fetch passerà a quella di decodifica.

ESECUZIONE DEL DATA HAZARD

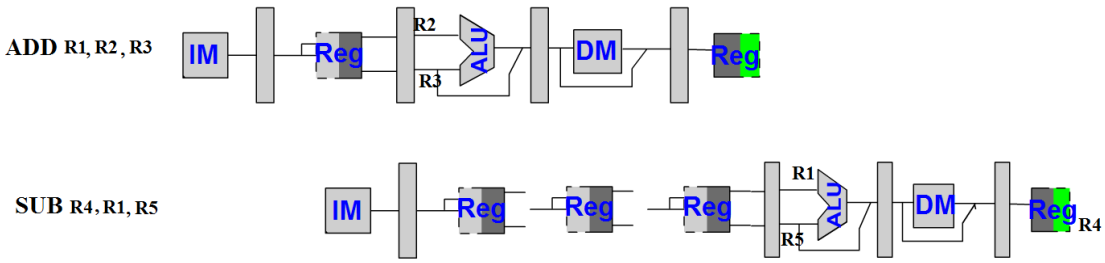
Quando c'è dipendenza dei dati tra due istruzioni, una soluzione per fare in modo che i risultati siano correti è l'introduzione degli stalli.

- Esecuzione scorretta del data hazard



Nella figura si vede che nello stesso ciclo di clock in cui l'istruzione add sta eseguendo l'operazione di ALU per produrre il risultato in R1, l'istruzione di sub necessita del registro R1 per eseguire la propria operazione di ALU. Quindi, verrà letto un valore errato e l'istruzione di sub restituirà un valore sbagliato.

- Esecuzione corretta del data hazard



Nella figura si vede che nel momento in cui l'istruzione di add esegue l'ALU, l'istruzione di sub necessita del registro R1. Quindi, per ottenere il valore di R1 corretto, la sub va in stallo per due cicli di clock e dopo di che procede con l'esecuzione della propria ALU, ottenendo così un risultato

corretto.

NUMERO DI CICLI DI CLOCK

Il numero di cicli di clock di stallo che vanno inseriti dipende dalla distanza tra le due istruzioni. Chiaramente, inserendo gli stalli, ci discostiamo dalla condizione ideale per cui a ogni clock eseguiamo un istruzione. Per la pipe però ogni ciclo di clock corrisponderà all'esecuzione di un istruzione, considerando anche l'esecuzione delle nop, nonostante si tratti di un istruzione inutile. L'introduzione degli stalli inoltre, aumenterà il CPI che non sarà più uguale a 1, bensì sarà maggiore di 1. Ecco 2 esempi:

add r1, r2, r3	IF	ID	EX	MEM	WB			
sub r4, r1, r2		IF	ID	***stall***	***stall***	EX	MEM	WB

subi r3, r2, 10	IF	ID	EX	MEM	WB			
add r1, r2, r3		IF	ID	EX	MEM	WB		
addi r4, r1, 5			IF	ID	***stall***	EX	MEM	WB

Nell'esempio, si vede come, nel primo caso, per due istruzioni a distanza 1, si renda necessario usare 2 cicli di clock di stallo. Invece, nel secondo caso, abbiamo una dipendenza dei dati tra due istruzioni che hanno distanza pari a 2, perciò si usa un solo ciclo di clock di stallo.

DATA HAZARD: INTRODUZIONE DEGLI STALLI

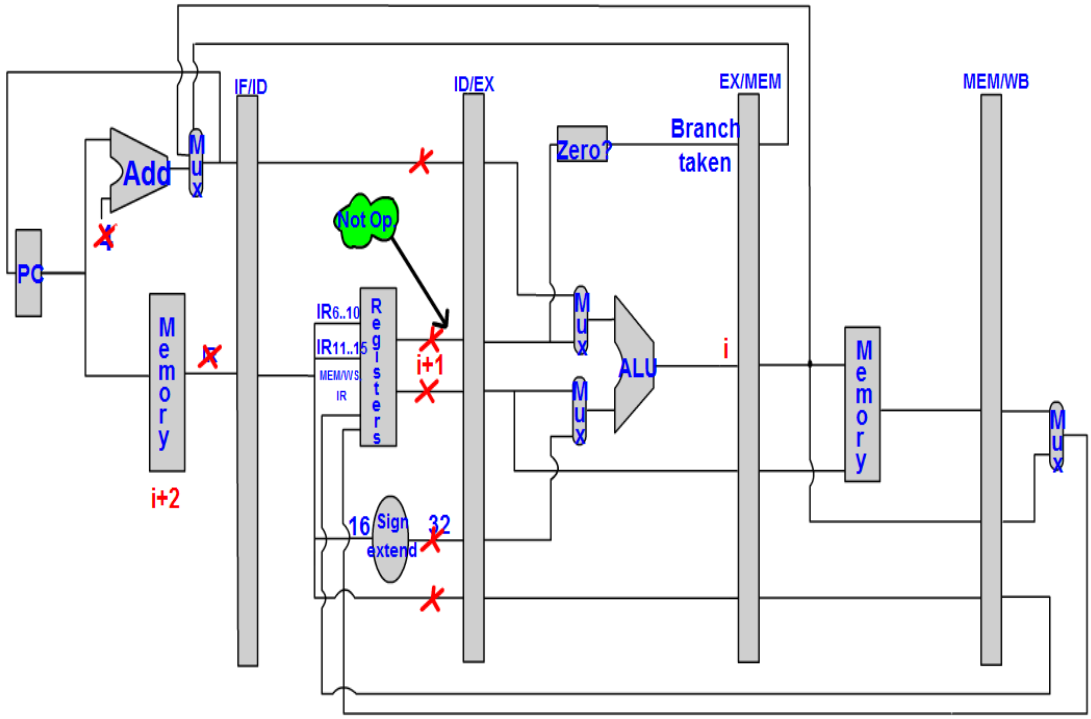
Poichè il data hazard viene scoperto nella fase ID, quando viene introdotto uno stallo per i data hazard:

• Viene bloccata l'istruzione nella fase ID impedendo l'aggiornamento del registro IF/ID;

• Viene bloccata l'istruzione nella fase IF non aggiornando il PC, altrimenti l'istruzione va nella fase di ID e sovrascrive l'istruzione che è già lì;

• Vengono scritti sul registro ID/EX i segnali di controllo relativi a una nop (non serve passare il codice operativo della not op, poichè tutti i segnali di controllo relativi a EX, MEM, WB sono contenuti in ID/EX). Fisicamente, possiamo immaginare che all'uscita del IF/ID vi sia collegata un'unita di controllo che fornisce i dati d'ingresso a un MUX, il quale verrà condizionato dagli eventuali stalli in modo da produrre in uscita sul ID/EX. Tale uscita sarà una nop (se c'è dipendenza dati) oppure sarà un segnale (se non c'è dipendenza dati);

I cicli di stallo vengono ripetuti fino a quando non viene aggiornato il registro destinazione.



Per disabilitare la scrittura in memoria (come nel caso delle not op) è necessario che vi sia un segnale di controllo che possa disabilitare la scrittura (ad esempio, regwrite=0). Lo stato del processore rimarrà quindi invariato.
Per sapere se introdurre o meno lo stallo e quanti sono gli stalli da introdurre (0 , 1 o 2) vi è un apposito hw che verifica le varie dipendenze tra le istruzioni.
Esempio:

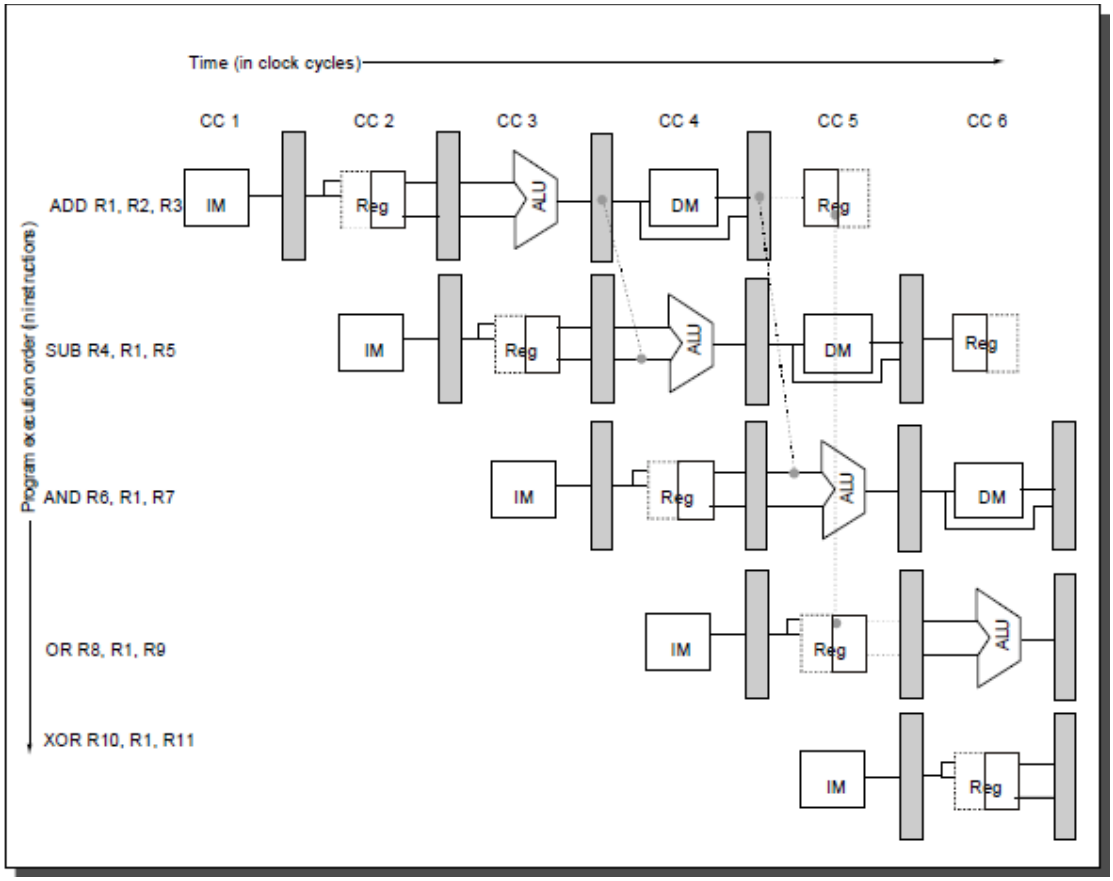
ID	EX	MEM
rs, rt	Rd, Rt	Rd
	regwrite=1	regwrite=1

rs, rt: registri riguardanti le istruzioni di tipo R
Rd (EX): registro per le istruzioni di tipo R
Rt: registro per le istruzioni di tipo I
Rd (MEM): registro destinazione
Se i registri di ID coincidono con i registri di EX o MEM e questa istruzione va ad aggiornare il banco di registri, allora c'è una dipendenza dati. In questo caso si realizza lo stallo. Chi controlla il verificarsi della dipendenza? Un apposito modulo hw che come ingressi ha rs e rt, i Rd e il regwrite. Mediante questi ingressi riesce a capire se serve introdurre uno stallo.

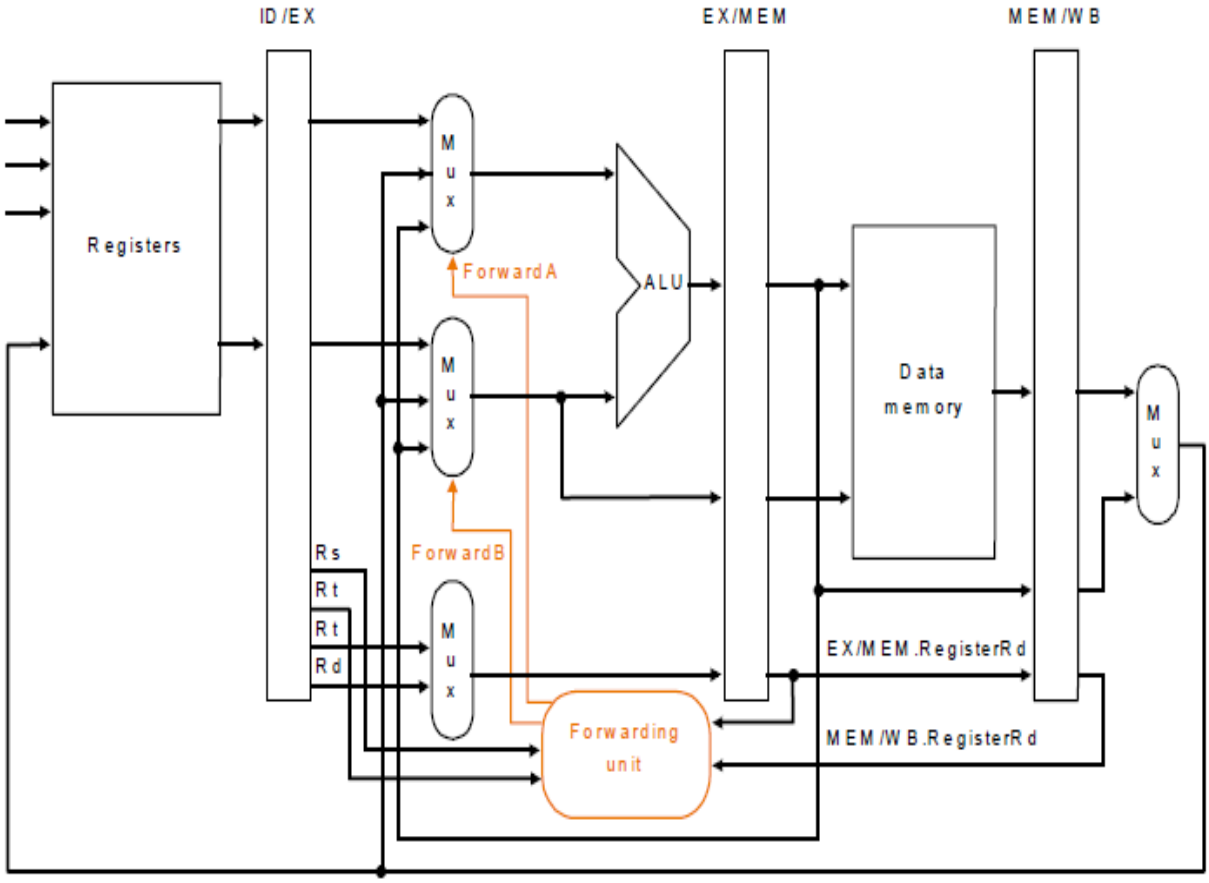
Esempio 2:
Se c'è una dipendenza tra ID e EX:
ID | EX
nel ciclo di clock successivo l'istruzione in ID si blocca mentre quella in EX viene eseguita e passata alla MEM, lasciando una nop (regwrite=0 in EX) nella fase di EX:
ID | Nop | MEM
A questo punto ci sarà dipendenza tra ID e MEM, quindi nel ciclo di clock successivo l'istruzione in

ID rimarrà ancora bloccata mentre quella nel MEM passa al WB, lasciando una nop (regwrite=0 in EX e MEM) nel MEM:
ID | Nop | Nop | WB
Al prossimo ciclo di clock, non vi saranno più dipendenze che vincolano l'istruzione in fase di decodifica, quindi questa potrà progredire alla fase di EX, mentre i Nop andranno in fase di MEM e WB e sfileranno progressivamente alle fasi successivi seguendo i cicli di clock.

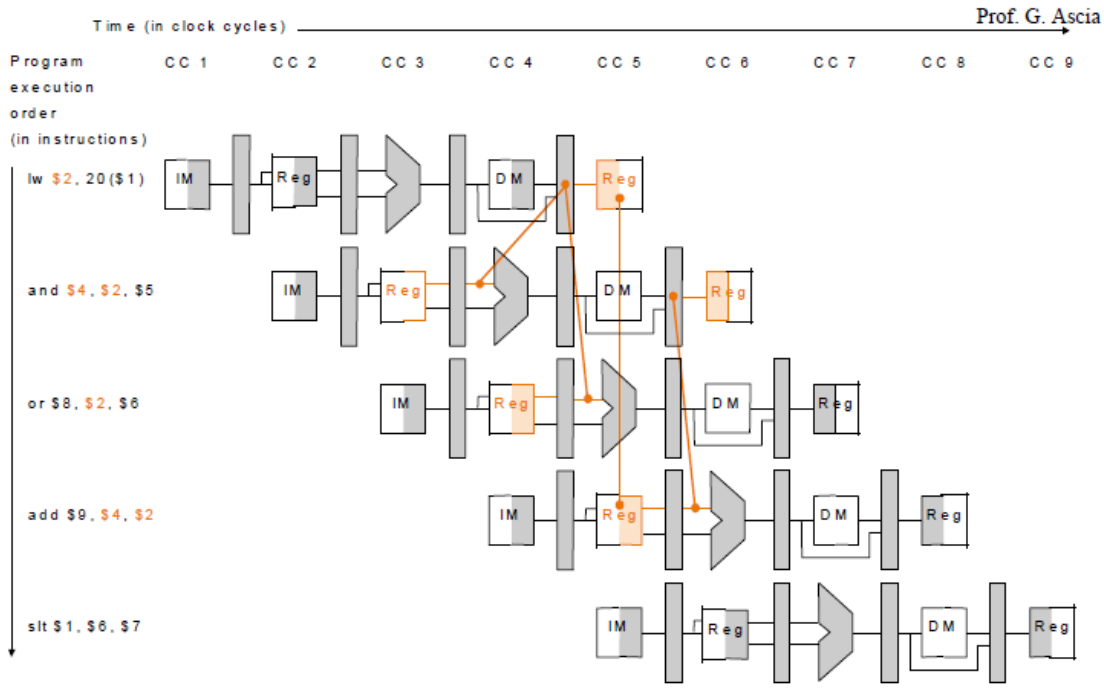
DATA HAZARD: FORWARDING
Anzichè aspettare che il dato sia scritto nei registri, noi potremmo andare a utilizzare il dato non appena è disponibile.
Esempio:



In figura possiamo vedere come il risultato prodotto dall'ALU della prima istruzione venga passato al registro EX/MEM. Da questo registro, passa immediatamente all'ingresso dell'ALU dell'istruzione seguente, senza attendere che venga scritta nel banco dei registri.
Se invece, guardiamo la terza istruzione, vediamo che l'ALU prende il valore che le occorre direttamente dal registro MEM/WB.
Da qui, possiamo vedere come venga sempre sfruttato il registro EX/MEM se tra le operazioni dipendenti vi è distanza 1, mentre se tra le operazioni dipendenti vi è distanza 2 si sfrutta il registro MEM/WB. Inoltre, è chiaro che questo sistema fa risparmiare i cicli di clock che con l'uso degli stalli andrebbero persi.
L'implementazione hw è però più complessa.



Agli ingressi dell'ALU abbiamo le uscite di due MUX. Questi MUX hanno agli ingressi: l'uscita del registro ID/EX (ossia il dato che è stato letto in fase di decodifica), valore che utilizziamo in assenza di dipendenza dati; l'uscita del WB i cui valori sono: o un dato che è stato scritto durante l'accesso alla memoria, oppure un dato di tipo R (tale uscita verrà usata se la distanza tra le istruzioni dipendenti è 2); l'uscita di EX/MEM che verrà sfruttutta nel caso in cui vi sia dipendenza tra istruzioni consecutive (distanza 1).
La forwarding unit ha il compito di generare i segnali di controllo (ForwardA, ForwardB) per il MUX e quindi dovrà verificare se vi sono dipendenze dati tra le istruzioni.
Gli ingressi della Forwarding unit sono: Rs e Rt che sono i registri sorgenti dell'istruzione che si trova nella fase di execute; il registro destinazione dell'istruzione che si trova nella fase di memory access; il registro destinazione dell'istruzione che si trova nella fase di writeback.
Sopra non sono disegnati, ma bisognerebbe inserire altri due ingressi che dovrebbero essere i segnali di regwrite della fase di memory access e del writeback, poichè la dipendenza dati potrebbe verificarsi soltanto se i regwrite sono settati a 1, altrimenti verrebbero ignorati e la dipendenza non sarebbe mai verificata.
Come si vede, la complicazione hw deriva dai MUX in ingresso all'ALU e dalla Forwarding Unit.
La forwarding unit è una rete puramente combinatoria poichè produce le uscite unicamente in funzione degli ingressi.
La presenza di questi componenti può avere un effetto negativo sulle prestazioni. Se ad esempio, la fase di Ex fosse la più lenta della pipe, l'introduzione dei MUX peggiorerebbe ulteriormente la situazione.
Grazie alla forwarding unit eliminiamo gli stalli in quasi tutti i casi. Fa eccezione il caso della loadword.



Nella figura sopra si vede come la dipendenza tra la LW e la AND non permette il forwarding, poichè il dato utile all'AND è disponibile solo dopo essere passato in memoria, ossia alla fine del quarto ciclo di clock. Però all'AND serve già all'inizio del terzo. Quindi, è comunque necessario introdurre uno stallo. Ma il forwarding è un approccio comunque più vantaggioso di quello che usa solamente stalli.

SOFTWARE SCHEDULING

La soluzione degli stalli e quella del forwarding è di tipo hw. Il software scheduling è una soluzione di tipo software in quanto chiede al compilatore la scelta ottimale della sequenza di istruzioni. Per ottimale si intende che il compilatore deve cercare di ridurre al minimo combinazioni di istruzioni tali da creare dipendenza dati.
Esempio:

Try producing fast code for

a = b + c;

d = e – f;

assuming a, b, c, d ,e, and f in memory.

Slow code:

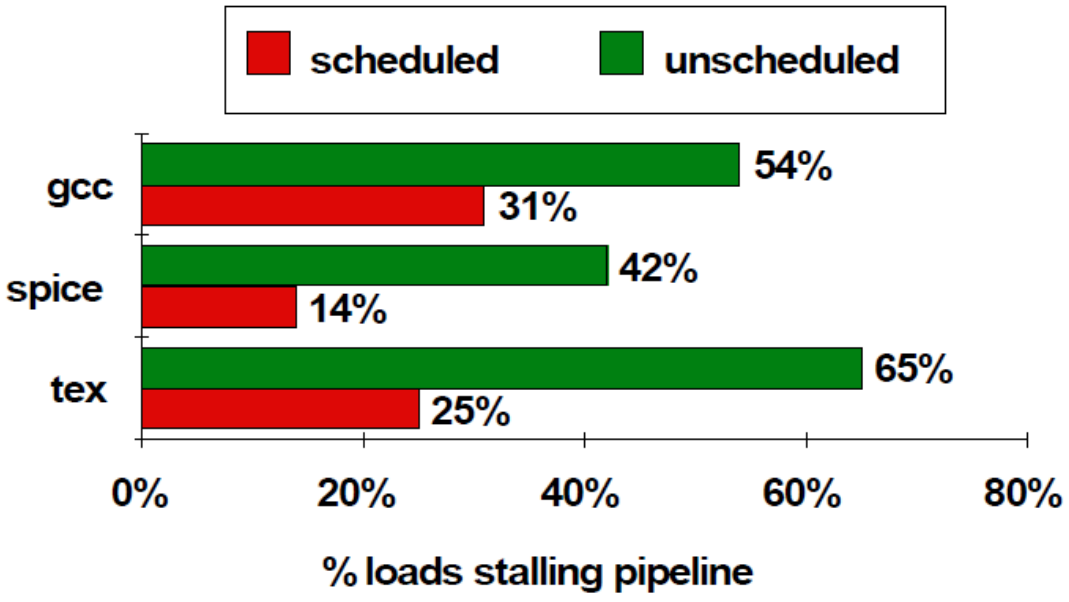
```
LW    Rb,b
LW    Rc,c
ADD   Ra,Rb,Rc
SW    a,Ra
LW    Re,e
LW    Rf,f
SUB   Rd,Re,Rf
SW    d,Rd
```

Fast code:

```
LW    Rb,b
LW    Rc,c
LW    Re,e
ADD   Ra,Rb,Rc
LW    Rf,f
SW    a,Ra
SUB   Rd,Re,Rf
SW    d,Rd
```

Nello slow code vediamo come la terza istruzione dipenda dalla prima (introduzione di due stalli) e dalla seconda istruzione(introduzione di uno stallo) con la tecnica degli stalli. Con il forwarding avremmo dipendenza tra la seconda e la terza e dovremmo introdurre uno stallo. La stessa situazione si ripete con la settima istruzione che dipende dalla quinta e dalla sesta.
Per realizzare il Fast code dobbiamo tenere presente che in alcuni casi, gli ordini non possono essere invertiti: ad esempio le load di Rb e Rc non potrebbero mai andare dopo l'ADD. Vediamo come con la tecnica degli stalli si avrebbero complessivamente 2 stalli, mentre col forwarding non se ne avrebbe nessuno. Se non vi fosse alcuna forma di gestione hw, dovremmo inserire delle not op di tipo software per evitare di ottenere risultati errati.

PERCENTUALI DI STALLI
Se utilizziamo un tipo di schedulazione ottimale otteniamo un numero di stalli che in percentuale è inferiore rispetto al caso in cui non scheduliamo.



BRANCH

Quando eseguiamo un'istruzione di branch, noi conosceremo l'esito del branch solo nella fase di accesso alla memoria. Quando arriva l'istruzione alla fase di accesso alla memoria, teoricamente potrebbero entrare nella pipe tre istruzioni.

Quando andiamo a scoprire l'esito del branch abbiamo due possibilità: il branch non è preso oppure no. Se è preso le istruzioni nella pipe sono buone e quindi utilizzabili. Se il branch è preso, significa che le istruzioni che si trovano nella pipe in fase di IF, ID e EX non sono buone e quindi non devono essere completate così da non alterare lo stato del processore.

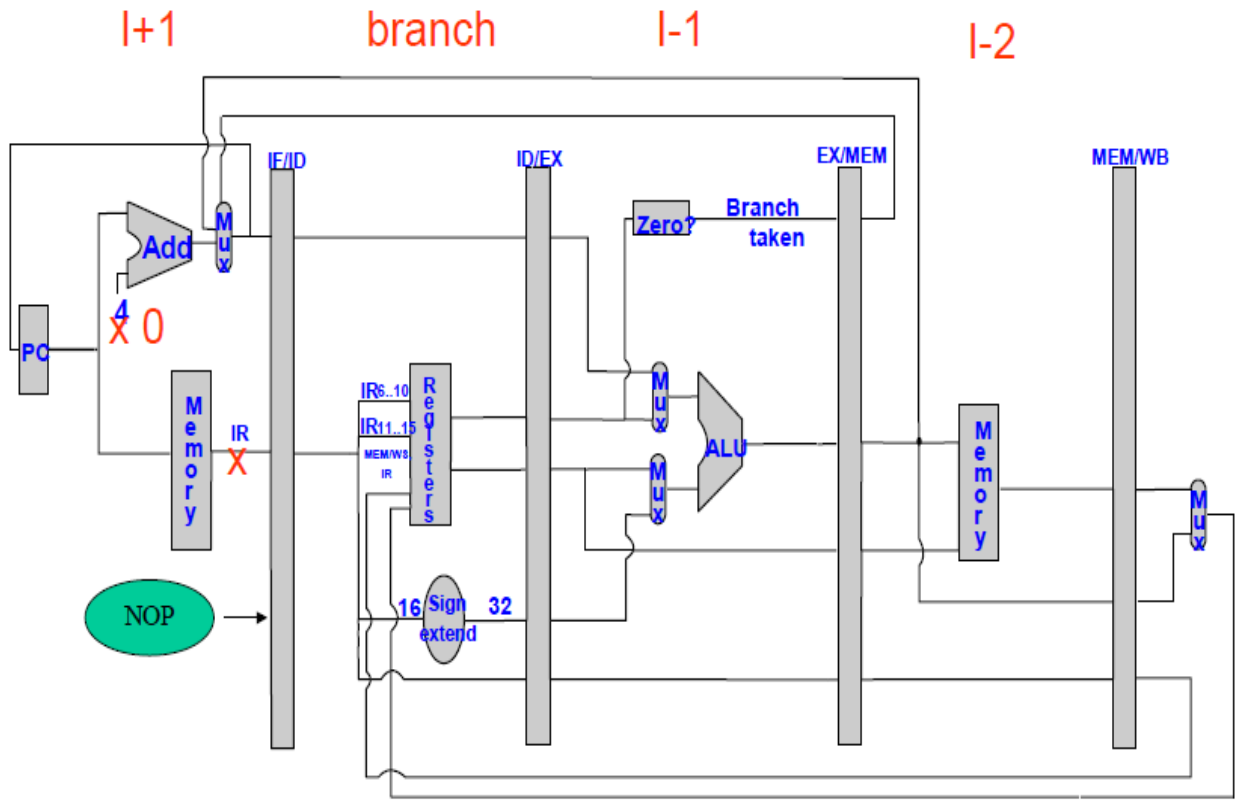
Per gestire questa situazione, abbiamo diverse soluzioni.

Una soluzione potrebbe essere l'introduzione degli stalli. Non appena scopriamo che l'istruzione che stiamo eseguendo è un'istruzione di branch (ancora non sappiamo se è presa oppure no) nella fase di ID, bloccheremo le istruzioni successive nella fase di fetch, ossia impediamo che le istruzioni di IF passino in fase di ID finchè il branch non esce dal MEM. Quindi il branch passa avanti finchè non arriva al MEM. Al termine della fase di accesso alla memoria si verifica se è preso oppure no. Se non è preso, il PC conterrà l'indirizzo dell'istruzione successiva a quella di branch. Se è preso, il PC conterrà il nuovo indirizzo, quello del salto. Nel ciclo di clock successivo andremo ad effettuare il IF vero e proprio oppure l'istruzione che è all'indirizzo di salto. Questa soluzione è semplice da realizzare dal punto di vista hw, ma è devastante per le prestazione perchè per ogni branch bisogna introdurre tre cicli di clock di stallo che hanno sicuramente un peso significativo.

Una soluzione utile a migliorare le prestazioni sarebbe tentare di calcolare l'esito del branch e l'indirizzo target nella fase di ID, anticipando ciò che normalmente avviene nella fase di MEM. Questa soluzione è negativa sul piano hw perchè dovremmo introdurre un sommatore che calcola il target e un altro blocco che valuti la predizione. C'è anche il vantaggio che dovremmo introdurre un solo ciclo di clock di stallo, anzichè tre.

Questo significa che nella fase di EX del branch andremo a fare anche il IF dell'istruzione successiva o dell'indirizzo target.

Concretamente, introdurre un ciclo di clock di stallo significa bloccare il PC e impedire che venga sovrascritto il contenuto di IF/ID, inviando una nop.



Al termine di questo ciclo di clock, la nop avanzerà come nei casi visti precedentemente.

FREQUENZA BRANCH
I branch con i salti in avanti si verificano più frequentemente rispetto ai branch dove i salti sono all'indietro o incondizionati. Di conseguenza si verificano più salti in avanti e meno salti all'indietro.

QUATTRO SOLUZIONI PER I BRANCH
1) Si attende che il branch venga esaminato a seguito della fase di MEM per vedere se è preso oppure no. Nel frattempo, nelle altre fasi si impongono i 3 stalli.
2)Predict Branch Not Taken. Si presume che il branch non sia preso, quindi le istruzioni successive vengono fatte entrare nella pipe, quindi, questo vuol dire, che se effettivamente il branch non è preso, avremo guadagnato tre cicli di clock. Tuttavia, la percentuale di branch non preso è del 47%. Se invece, il branch viene preso bisognerà cancellare le istruzioni in fase di IF, ID e EX (nello specifico all'interno dei registri) che si trovano all'interno della pipe sovrascrivendole con le nop. Se il branch viene anticipato nella fase di ID (caso del predict untaken), l'unica istruzione che andremo a cancellare sarà quella nella fase di IF, ma ci sarà anche un'eventuale guadagno inferiore nel caso il branch sia effettivamente non preso.

3) Predict branch Taken. Si presume che il branch sia preso. Nella fase di IF dovremmo già conoscere qual è l'indirizzo target dell'istruzione successiva a quella di branch. Quindi in modo anticipato, noi dovremmo andare a fare il IF dell'istruzione che si trova all'indirizzo target. Ma questa soluzione non è applicabile perchè effettivamente noi conosceremo l'indirizzo target nella fase di ID, quindi se volessimo utilizzare questa soluzione dovremmo aggiungere un ciclo di stallo obbligatorio. Fa eccezione il raro caso in cui si abbia già a disposizione una tabella con i possibili target, ma si tratta di roba avanzata.

lezione 6 giugno 2011

(slide 45) Analizziamo l'ultimo modo di gestire il branch che chiamiamo Delayed Branch. Questa tecnica consiste nell'andare a piazzare dopo l'istruzione di branch delle istruzioni, una o più (dipende dalla fase in cui scopriamo l'esito del branch), che comunque verranno eseguite(sia che il branch venga preso o meno).Se l'esito del branch lo abbiamo nella fase di codifica allora l'istruzione da piazzare sarà una sola.In questo caso la risoluzione del problema viene spostata da hardware a software.Mentre ricordiamo che via hardware avevamo fatto in modo di inserire delle NOP finchè non era noto il risultato o per esempio nel PREDICT UNTOKEN si facevano entrare una serie di istruzioni e se veniva preso tutte le istruzioni venivano portate a NOP.Qui l'hardware non si preoccupa delle istruzioni successive, se ne occupa quindi il software.Il compilatore si occupa del problema facendo in modo che subito dopo il branch vengano eseguite istruzioni che comunque anche se eseguite non creino danno al sistema, ottenendo il medesimo risultato finale. Se tale istruzione non dovrebbe essere trovata allora il software provvederà ad andare a mettere un NOP naturalmente via software.Il problema a questo punto è: quale sarà l'istruzione da mettere dopo il branch?
(per spiegarlo faremo l'esempio nella fase di decodifica).
(slide 46)In questa slide possiamo notare come l'andamento del procedimento del delayed branch da lo stesso risultato sia nel caso in cui il branch venga preso sia in cui non venga preso.
(slide 47)In questa slide vediamo alcune soluzioni di come noi potremmo andare a scegliere l'istruzione consecutiva al branch.
"soluzione From Before": in questo caso notiamo che l'istruzione del branch(if R2=0) con l'istruzione immediatamente precedente a questa(ADD R1,R2,R3) non hanno nessun elemento in comune.Quindi piuttosto che metterla prima del branch la metto dopo il branch scambiando l'ordine tanto, come si diceva, non sono legate. Con questa operazione non altero il comportamento del programma in quanto il risultato finale come si può vedere nel riquadro immediatamente sottostante nella slide è uguale. In sostanza vado a mettere nel Delay Slot l'istruzione immediatamente precedente.Quindi abbiamo un istruzione utile eseguita dopo il branch sia nel caso il branch sia preso sia in cui non sia preso.

"soluzione From Target": Questa soluzione viene usata se abbiamo dei salti all'indietro. In questo caso abbiamo un'istruzione SUB che rappresenta l'istruzione a cui si va quando viene eseguito il salto all'indietro. Notiamo che prima dell'istruzione del branch(if R1=0) abbiamo un'istruzione(ADD R1,R2,R3) che dipende dall'istruzione del branch quindi non posso usare la soluzione "from before".Quello che si fa è andare a "copiare" l'istruzione SUB(quindi l'istruzione indirizzata dal target) dentro il Delay Slot, facendo caso che a Branch preso sarà comunque eseguita.In questo caso l'istruzione SUB la eseguiremo due volte, la prima quando ci tocca la seconda dopo il salto all'indietro. Adesso vediamo cosa accade dopo questo inserimento nel riquadro inferiore(quello puntato dalla freccia Become s). Notiamo che a questo punto il salto non viene fatto alla funzione SUB(come prima) bensì all'istruzione immediatamente successiva, questo perchè vogliamo che alla fine si abbiamo come sopra che il sub si eseguito solo due volte. Per fare ciò si setta il target= target + 4 byte(se 4 byte è la grandezza di un istruzione). Cosa cambia sostanzialmente tra sopra e sotto? cambia il fatto che sopra, se il branch non viene preso la funzione SUB viene eseguita n-1 volte(la n-esima volta non viene eseguito), mentre nel riquadro sottostante la SUB viene eseguita tutte le n-volte(anche l'ultima volta). Cosa importantissima è che l'istruzione che va nel delay slot sia un istruzione che il suo risultato non venga usato in seguito senza cambiare il risultato finale. (esempio alla lavagna: supponiamo per esempio di avere dopo quella SUB come istruzione successiva: ADD R3,R4,R8.
In questo caso questa sub non potrebbe essere messa lì perchè il valore di R4 dovrebbe essere non quello ma quello relativo all'ultima volta che è stato preso.Nel caso in cui il branch è preso e nel caso in cui non è preso. Nella forma sotto il valore di R4 potrebbe essere diverso da quello che si ha sopra. Se ciò è possibile allora chiaramente se poi R4 viene utilizzato successivamente questa istruzione non è utile, se invece il comportamento è uguale sopra e sotto va bene questa soluzione. Quindi la condizione necessaria affinche la SUB sia messa nel delay slot è che R4,R5,R6 non siano usati all'interno del branch, cosicchè nella slide la parte sopra e sotto della colonna relativa al From Target abbino stesso stato finale.
"From fal through":qui abbiamo un salto in avanti. Anche qui non possiamo mettere l'istruzione prima del salto perchè c'è dipendenza. Cosa facciamo? andiamo a mettere nel delay slot l'istruzione prima dell'indirizzo target.Che è la SUB nel nostro esempio(istruzione tra branch e target). Anche qui si può fare ciò se la SUB modificando il risultato di R4 non altera il risultato finale. Se dopo la SUB ad esempio c'è una memorizzazione e su R4 scriviamo altro. In questo caso la mettiamo perchè non ha influenza sul risultato finale. Se invece a branch preso tale istruzione altera il resto del programma non può essere messa. Se il salto non è preso avremo la stessa cosa, se è presa teoricamente questa istruzione non dovrebbe essere eseguita. Se R4 mi servirà dopo non potrò metterlo e dovrò mettere delle NOP.
Considerazioni: nell'ultimo caso nel caso in cui non sia presa la condizione del salto vediamo che non cambia niente tra mettere la NOP o fare il From Fall throught. Quindi che senso ha? noi non sappiamo se le condizioni verranno verificate, ma facciamo in modo che se il branch è preso faciamo qualcosa di utile. Naturalmente piu avanti andiamo a mettere l'esito del branch più istruzioni dovremmo mettere negli slot(anziché una , ne metto 2,3). Più vado avanti quindi piu difficile è trovare istruzioni che non creano danno.
(slide 48 l'ha saltata)
(slide 49)Ora che abbiamo visto le varie soluzioni "ideali" di come gestire il branch, andiamo ad analizzare cosa effettivamente accade nel modello "reale".Le scorse volte abbiamo visto come in un pipeline a N stati lo speed up era uguale al numero di stati. Invece in questo caso abbiamo che lo speed up è:

SPEEDUP= CPU TIME SEQUENZIALE/CPU TIME PIPELINE (questo rapporto lo possiamo esprimere come il rapporto tra CPI SEQUENZIALE* PERIODO DI CLOCK SEQUENZIALE/ CPI PIPE *PERIODO DI CLOCK PIPE).

Ora immaginando che gli stati siano perfettamente bilanciati(è più facile che non lo siano) e che l'introduzione dei registri tra uno stadio e l'altro non produca un overhead (ulteriore latenza) allora si avrebbe che Tck SEQ=TckPIPE(qui si mette bene in evidenza come il reale sia diverso dall'ideale in quanto tutte le condizioni elencate precedentemente affinchè i due T coincidano sono

difficili da realizzare). Immaginiamo che il periodo di clock sequenziale quindi sia uguale a quello pipeline,ovvero consideriamo il caso ideale.
In questo caso andando a semplificare nella prima equazione TckSEQ e TckPipe e consideranco che CPIpipe=CPIideale + NR(SOMMA DI CPI IDEALE(uguale a 1 in una situazione ideale dove non ho hazard) + NR(numero di cilci di stallo per istruzione, dovuti a hazard, controhazard e struct hazard)), otteniamo che lo speed UP= CPI SEQ / (1+ NR).
Poi abbiamo visto che CPISEQ = STADI DELLA PIPE.
(slide 50) vediamo degli esempi numerici che ci danno l'idea dello spostamento rispetto al caso ideale. IMMAGINIAMO DI ESEGUIRE UN PROGRAMMA E DI SCOPRIRE CHE IL 20% DELLE ISTRUZIONI SONO DI LOAD, 20% DI STORE,10% DI BRANCH E 50% DI ALU.QUI ABBIAMO UN UNICA MEMORIA(quindi hazard strutturale). Utilizzo il forwarding e so che il 50% delle load è seguita da un istruzione che dipende da essa(quindi anche se uso il forwarding nel 50% dei casi dobbiamo introdurre un ciclo di stallo). In questo caso dovremmo tenere conto di questa dipendenza dati.Inoltre calcolo l'esito del branch nello stadio di Decodifica.

Calcolo del CPI

CPIpipe=CPI IDEALE+ NR= (sommatoria delle frequenze degli hazard per il numero di cicli di clock di stallo richiesti per quell'hazard)+(la frequenza del brench + il numero dei cicli di stallo per quel branch) + (frequenza della slot + numero di cicli di stallo per quella slot).

A QUESTO PUNTO IL PROBLEMA È SAPERE LE FREQUENZE E I NUMERI DEI CICLI DI STALLO.
Per quanto riguarda l'hazard strutturale abbiamo detto che abbiamo un unica memoria.In questo caso abbiamo un conflitto tra le istruzioni in fetch e quelle che devono leggere un dato. Solo quando eseguiamo una load o una store avremo una contesa per la memoria. Nel nostro esempio la contesa + 20+10% =30%(somma delle frequenze di store e load). In questo caso abbiamo un unico ciclo di clock perchè dobbiamo aspettare che load e store accedano alla memoria e dopo effettuiamo il fetch. Per quanto riguarda il brench non specifichiamo la tecnica, immaginiamo quella degli stalli, poiché calcolo l'esito nella fase di decode quando cicli di stallo posso introdurre? Uno! E la frequenza? è data nei dati del problema ed è 20%. Per quanto riguardo l'hazard ho il dato del 50%, quindi come frequenza devo prendere il 50% del 20% quindi 10% e inoltre un ciclo di stallo perchè stiamo utilizzando il forward.
(slide 51) sostituendo i numeri ottengo i valori della slide . SpeedUP= 3.125 inoltre notiamo che il CPI REALE non è uno ma 1,6.
(slide 52) qui abbiamo un altro esempio. Qui a differenza di prima abbiamo 2 memorie e non utilizzando il forwarding abbiamo solo stalli. Poi l'esito del branch è nella fase di accesso alla memoria. Poi abbiamo altri dati che ci serviranno nei calcoli. Al solito per calcolare lo speed up, qui hazard strutturali non ne abbiamo, abbiamo branch e data hazard tra istruzioni di distanza 1 e tra istruzioni di distanza 2(qui infatti dobbiamo distinguere, nel primo caso si inseriscono 2 cicli di stallo nell'altro caso uno solo). Per quanto riguarda gli stalli nel caso del branch poiché siamo in accesso a memoria sono 3 i cicli di stallo.Le frequenze del breanch c'è l ho nel dato mentre per i dati dell'hazard ho il 30% del 40% per distanza 1 mentre il 10% del 40% per distanza 2.
(slide 53) numericamente il risultato viene 2,66.
Per quanto riguarda il controllo se utilizzassimo la tecnica del predict untoken si dovrebbe anche dire nei dati con quale frequenza indoviniamo l'esito del branch.

SOTTOSISTEMA DI MEMORIA

ALTRO SET DI SLIDE SOTTOSISTEMA DI MEMORIA

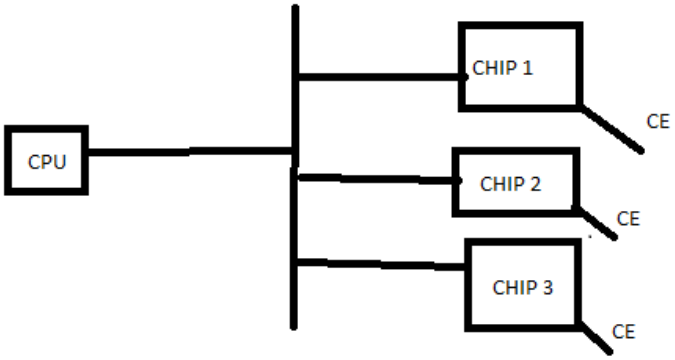
(slide 1) Visti i tempi limitati diremo qualcosina sulle varie tipologie di memoria e per quanto riguarda le memorie cache faremo un breve cenno perchè le studieremo nel corso di calcolatori.

(slide 2) la memoria a cui faremo riferimento è la RAM. Memoria di accesso diretto che possiamo distinguere in memoria statica e dinamica. Quella che tipicamente chiamiamo principale è memoria fatta con ram DINAMICA. La statica viene utilizzata per la cache. Vedremo perchè è utile utilizzare la cache e non soltanto ram dinamica(la cache sarà utile per dare l'illusione di avere l'intera memoria sia fatta di ram statica). Con buona approssimazione potrà accadere questo. La cache consuma molto di più e costa di più della ram dinamica per questo motivo viene usata poco. Ram dinamica 4 gb e ram statica (cache) 4 mb. La potenza dissipata nel caso della ram statica è molto più elevata rispetto alla ram dinamica. La ram dinamica conviene perchè consuma molto di meno, costa di meno, possiamo cotruire memorie molto più grandi. (sostanzialmente possiamo usare dei condensatori). Nel caso della memoria statica un bit lo immagazziniamo come un sistema tipo quello del flip flop in SR. Conviene la statica perchè potreste lavorare con una velocità di esercizio paragonabile a quella del processore mentre quella dinamica necessità di più cicli di clock. Se abbiamo solo ram dinamica la fase di fetch non avviene in un micro ciclo di clock ma ne ha bisogno ti tanti! Anche se la fase di decode potrebbe essere fatta comunque in un microciclo di clock.
(slide 3)
"RAM STATICA". Tipicamente è una memoria caratterizzata da bassa densità bassa capacità per chip che contengono un numero elevato di bit.Per memorizzare ricordiamo che sono necessari 4 o 6 transistor. Tipicamente questo tipo di memoria viene usata per registri interni al processore o per memoria cache. Qui vedete uno schema di memoria statica. A sinistra abbiamo le linee di ingressi che arrivano la processore(A0...), un insieme di dati in alto a destra che sono in ingresso e uscita(D0...D7) e una serie di segnali che sono:
WE: write enable , mi serve per decidere accesso in scrittura o in lettura;
CE: chip enable abilita o meno l'accesso alla memoria);
OE: output enable serve ad abilitare o meno le uscite, se è disattivata le uscite sono staccate dal MEM).
(slide 4) in questo disegno fatto dal prof vediamo uno schema di alto livello. Vedete che abbiamo un decodificatore di righe e uno di colonna di colonna, dalla combinazione di righe e colonne centriamo il bit o l'insieme di bit (dipende dal chip) che verranno mandate in uscita all'amplificatore. Le uscite vengono amplificate e poi arriveranno in I/O di uscita. Nella fase di lettura nei circuiti di controllo vengono abilitati WE(1 lettura 0 scrittura) , CE viene messo a 1 e anche OE. Arrivano i bit da A1 ad A7(per le righe), da A8 a A15(per le colonne), viene memorizzato in ogni locazione un bit. Le decodifiche vengono abilitate dai segnali di ingresso settati prima. Dei buffer tristate(nel grafico sono messi a destra prima delle uscite) comandatati da OE mi mandano in uscita i dati.
(slide 5) vediamo cosa avviene in un ciclo di lettura(qui si suppone di lavorare con memoria asincrona). Dopo un certo tempo si abilita il CE che attivo a 0 naturalmente(ciò deve avvenire solo se l'indirizzo è stabile). Per quanto riguarda il WE qui rimane a 1 in questa slide costantemente. Dopo un certo intervallo Tacc da quando il CE è stato attivato, i dati saranno validi. In questo intervallo sulle linee di uscita(nella slide 4 è la linea perpendicolare prima delle I/O,) il dato è valido è potrebbe essere mandato in uscita sul bus dati per essere letto dalla CPU. Poi abbiamo il tempo da quando attiviamo l'OE a quando il dato è presente in uscita sui circuiti di memoria. Tipicamente il parametro che interessa per valutare le prestazione di ram statica è il tempo di accesso Tacc. Il parametro Trc(tempo di ciclo) è l'intervallo che trascorre tra un accesso alla memoria e il successivo. Come minimo tra un accesso e un'altro deve passare Trc. Nella statica Trc =Tacc quindi usiamo come indice Tacc. Nella dinamica è diverso. Le prestazioni si valutano non tanto dal Tacc ma il tempo di ciclo che risulta essere molto più alto rispetto al tempo di accesso. E dunque siccome due accessi consecutivi possono avvenire con una distanza temporale che come minimo è quella(immaginate il pipeline) qual'è il tempo minimo tra un fetch e il successivo? È limitato da Trc. non possiamo farne due in un t minore di Trc.
(slide 6) RAM DINAMICA. Per memorizzare un bit in una RAM dinamica sfruttiamo un condensatore ovvero si sfrutta la carica immagazzinata nella capacità parassita del circuito. In base alla differenza di potenziale diciamo se è 1 o 0. Perchè possa essere interpretato correttamente il

contenuto del condensatore si ha V superiore o minore di una certa soglia. C' è una circuiteria che va a spazzolare tutte le righe della ram dinamica e che rigenera i valori di questi condensatori.Il refresh è essenziale perchè essendo questi condensatori non ideali con il passare del tempo si scaricano e quindi hanno bisogno di essere ricaricati.Sicuramente abbiamo una complicazione per i refresh dei vari condensatori. Appositi circuiti di controllo di errore mi aiutano a vedere e anche a correggere l'errore. Qui abbiamo tempi di lettura decisamente più alti dalla memoria statica, non viene mandato l'intero indirizzo della CPU(se abbiamo 32 bit di indirizzamento li mando in due blocchi da 16). Esistono appositi segnali che servono alla memoria per dire che stiamo mandando prima l'indirizzo di riga e poi quello di colonna.
(slide 7) tali segnali sono il RAS (riga) e il CAS (colonna) che assomiglia alle CE nel senso che viene mandato l'indirizzo di riga(row address e si attiva il RAS dopo un certo tempo viene memorizzato l'indirizzo di riga poi viene mandato l'indirizzo di colonna(col address) e viene attivato il CAS. Tipicamente nelle Ram dinamiche prima si manda l'indirizzo di riga e poi di colonna.A questo punto considerando lo schema di slide 4 si può selezionare il bit o il chip richiesto. Qui vedete in queste righe due tipi diversi. Nel primo esempio l'output enable viene attivato prima dell'attivazione del CAS , in questo caso dopo un certo tempo dall'attivazione per un po' di tempo si invia della spazzatura(JUNK) sul bus dati ,poi per un certo tempo Tcac avremo il dato in uscita valido. Nel secondo caso l'output enable sarà attivato dopo il CAS. In quell'istante finale il dato sarà effettivamente disponibile e non avremo dei dati inutili prima dell'attivazione dell'OE ma avremo dei dati(non c'è la parte spazzatura JUNK e il tcac è piccolo. Trac corrisponde all'intervallo da quando è attivato il RAS a quando il dato è disponibile, questo è un parametro per valutare anche le prestazioni, poi c'è Tcac, da quando il dato è attivo a quando è realmente disponibile.
(slide 8) quali sono i valori per cui la CPU non deve aspettare la memoria. La cpu lavorerà al massimo senza aspettare la memoria. Nel caso di ram statica il Tacc<T (tempo che la cpu prevede per accedere alla memoria). T= N/f (n numero di accesso alla memoria e f frequenza). Noi abbiamo supposto N=1 ma potremmo avere anche implementazion i PIPELINE in cui una fase non è realizzata in un ciclo di clock ma in 2,3 ecc ecc. Nel caso di dinamica deve essere Trc < T. Pensate alla pipeline il fetch avviene ad ogni ciclo di lettura.
Nel caso in cui non soddisfiamo l'ultima condizione si ha che T=(N+Nw)/f>Tacc dove Nw sono i cicli di clock in più che ho. Finchè Nw è piccolo allora magari può essere accettabile l'attesa, ma se è troppo elevato non è possibile lavorare con la RAM dinamica.
(slide 9) saltata.

Lezione 8 giugno 2011

Introduzione argomento slide successive:
ritorniamo al problema dell'ultima volta. Noi abbiamo detto che un sistema di memorizzazione sarà caratterizzato da CPU e MEM. La memoria non è caratterizzata da un unico chip ma da un insieme di chip che messi insieme rappresentano il sistema. Ogni chip andrà a ricoprire uno spazio di indirizzamento della memoria. Cosa accade? Questi diversi chip potranno essere attivati o meno da un segnale CE attraverso quel bus dati sono collegati alla CPU tutti questi chip di memoria.

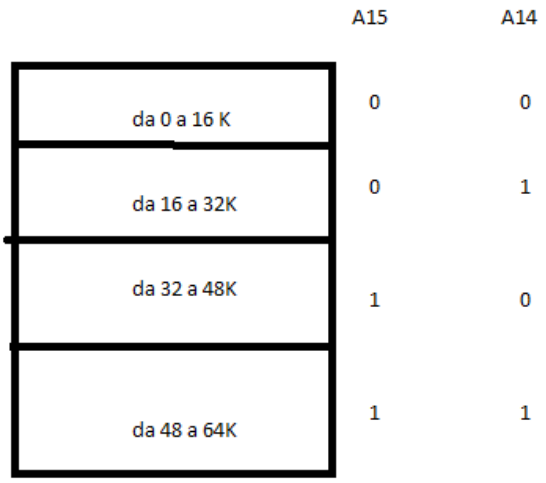


Ora cosa accade? Quando la CPU manda un indirizzo questo indirizzo viene mandato sostanzialmente a tutti quei chip contemporaneamente. Ma quell'indirizzo devo fare in modo che uno solo dei chip lo acquisisca, ovvero quello a cui è indirizzato realmente. Se supponiamo avere una memoria di 64K, se il primo indirizzo è tra 0 e 16K allora il chip 1 dovrebbe riconoscere l'indirizzo della CPU come indirizzo a se destinato e quindi questo indirizzo verrà utilizzato per fornire un dato in uscita, tutti gli altri capiranno che l'indirizzo non è rivolto a essi e non saranno attivati. Stessa cosa avverrà con il secondo chip se ci troviamo tra 16K e 32K. Per generare i CE abbiamo bisogno di un apposito circuito che dato un indirizzo proveniente dalla CPU generi i vari CE. Per la realizzazione della decodifica degli indirizzi e quindi per l'assegnazione di uno solo dei chip possiamo usare 2 soluzioni: GERARCHICA e LINEARE.
Se è gerarchica, noi abbiamo una cascata di decodificatori. Dato l'indirizzo della CPU questo andrà al decodificatore che produrra delle uscite. Di queste alcune saranno inviate direttamente ad un chip, altri invece segnali di controllo potrebbero essere usati per abilitare oppure meno dei decodificatori che a loro volta abiliteranno dei chip. E' gerarchica perchè abbiamo un decodificatore di primo livello a cui ne seguono altri di sotto livello.
L'altra è tale che esiste un unico decodificatore e quindi le uscite verranno tutte da li.

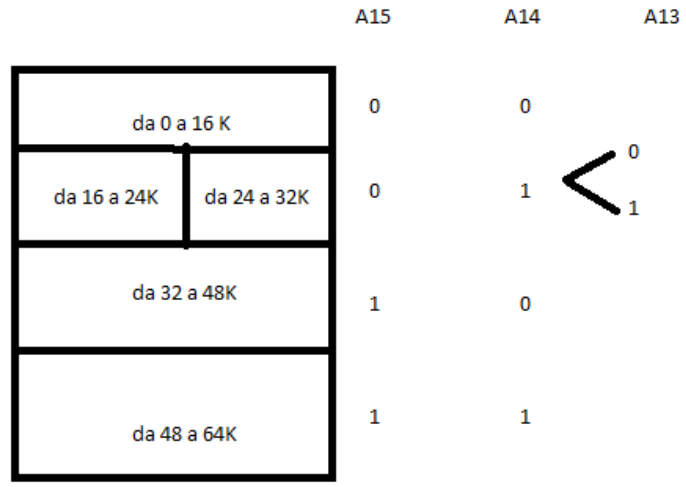
(slide 10) Qui vediamo un esempio di tipo Gerarchico. Abbiamo un decodificatore di primo livello a 4 uscite(perchè di fatto a determinare le uscite e il più grande dei chip presenti, in questo caso il più grande è 16k in una memoria di 64K quindi il numero di uscite sarà 64K/16K=4). Quindi numero di uscite in una tipologia gerarchica è uguale a:

uscitegerarchica= (memoria totale/chip di memoria più grande)

Per selezionare una di queste 4 uscite ci servono 2 bit che prendiamo dall'indirizzo generato dalla CPU, prendiamo noi i 2 bit più significativi , A15 e A14.



Le 4 uscite sono collegate direttamente ai CHIP enable. Ora noi abbiamo però per il secondo blocco da 16K quello che va da 16 a 32 composta da due da 8K.(analogalmente per il 3°blocco). La 4° regione da 48 a 64 in questo caso non riusciamo a coprirla perchè non abbiamo chip di memoria con cui posso mapparla in quanto nel nostro esempio arriviamo a coprire solo 48K. Il secondo blocco poiché è composto da due da 8 dobbiamo discriminare se arriva al primo chip o al secondo, per distinguerlo abbiamo bisogno di un decodificatore che ha 2 uscite che abilita uno o l'altro. In questo caso per distinguere le regioni 16/24 e 24/32 utilizziamo il bit 13, perchè i soli bit 14 e 15 non ci bastano.

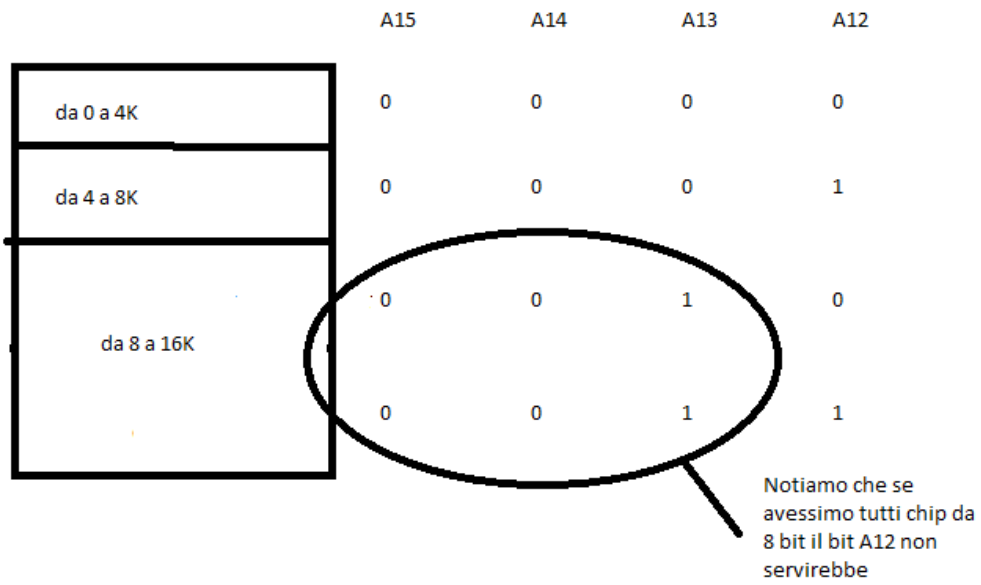


Stesso identico ragionamento possiamo fare per l'altra coppia di bit. Se immaginiamo che al posto dell'ultimo da 8 ne avessimo due da 4 allora l'uscita dovrebbe essere inviata prima ad un altro decodificatore che grazie ad A12 distinguerebbe i 2 chip da 4.

(slide 11)
qui abbiamo un altro esempio del modello gerarchico, ovvero il caso in cui abbiamo 3 chip da 8 K e 2 chip da 4K. In questo caso per mappare la memoria sui due chip abbiamo bisogno di un decodificatore che avrà 8 uscite(perchè 64K/8 che è la più grande quindi 8). I diversi chip saranno mappati sulle diverse uscite del primo decodificatore. Essendo 8 mi serviranno per distinguerle i bit 15A,14A,13A. Avendone 2 da 4K avremo un decodificatore secondario che con il 12A mi distingue.
(slide 12) qui vediamo il modello LINEARE. Abbiamo un unico decodificatore dove le uscite vanno a selezionare i chip p di memoria. In questo caso le uscite sono determinate dal chip più piccolo(64K/ 4 che è il più piccolo=16 uscite). (in altre slide si diceva che nel decodificatore abbiamo n entrate e 2^n uscite).

Uscitelineare=(memoriatotale/chip di memoria più piccolo)

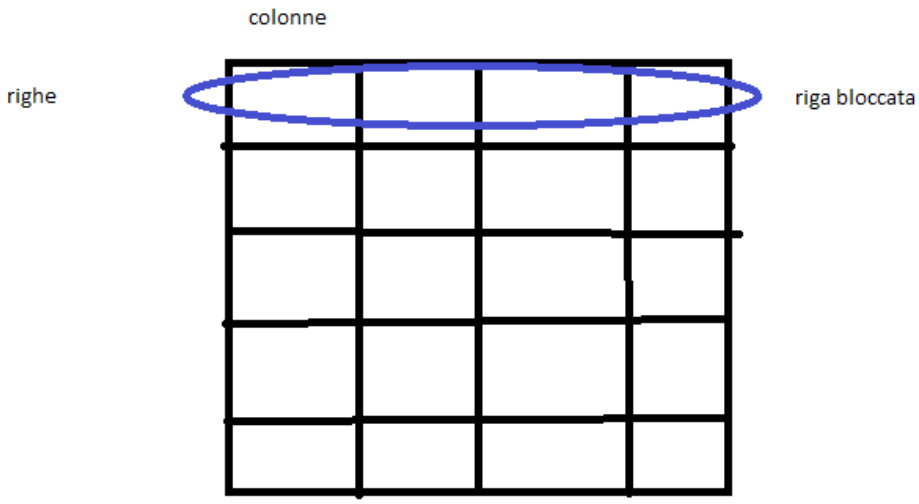
In questo caso infatti abbiamo 4 ingressi. In questo caso succede che nel caso dei chip da 4 K l'uscita viene utilizzata direttamente per selezionare il chip. Per quelli da 8K succede che in realtà utilizziamo 2 uscite buone che unite mi identificano quello da 8K. Se per esempio guardiamo le uscite 8K-12K e 12K-16K unendola copre l'intervallo di 8K ad 8K-16K che mi identificano un chip da 8K.
In questo caso ogni configurazione di quei 4 bit individua un chip da 4K, in questo caso 0010 e 0011 mi identificano il primo da 8K infatti se vediamo i primi tre bit sono per entrambi uguali quindi il quarto bit è inutile come d'altronde sarebbe inutile se avessimo una configurazione totale con soli chip da 8K.



(slide 13)Quando noi abbiamo word di memorie 16-32-64 purtroppo non abbiamo chip con quella sequenza di parallelismo ma abbiamo chip che danno in uscita un numero di bit più basso(1-2-4-8). Come costruiamo quindi la memoria con un certo numero di parallelismo? Banalmente basta mettere in parallelo 2 chip. Se la Cpu genera un indirizzo(nell'esempio A0-A7) viene inviato in parallelelo a entrambi i chip insieme al segnale di CE. Se vogliamo accedere a quella area di memoria entrambi sono attivi altrimenti no. Il parallelismo lo otteniamo avendo più chip messi in parallelo agenti sulla stessa area di memoria e facendo riferimento allo stesso indirizzo.
(slide 14) Riguardo le memorie dinamiche vediamo l'evoluzione che nell'ultimi anni ha avuto la RAM dinamica. Inizialmente era caratterizzata dal fatto che come abbiamo visto la scorsa lezione

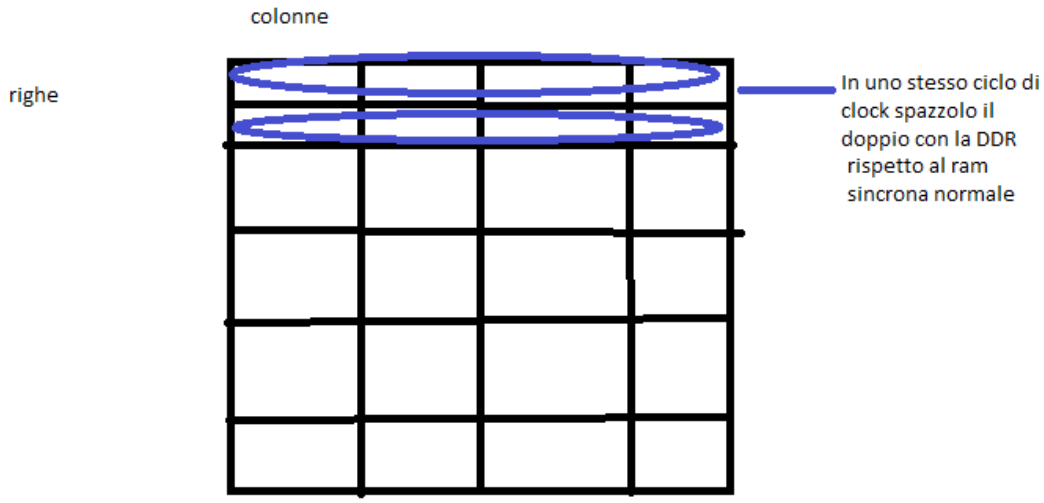
viene generato il RAS e l'indirizzo di riga viene acquisito, poi viene attivato CAS e dopo un certo tempo abbiamo il dato disponibile. Al termine di questa operazione si ricomincia da capo. Questo è l'approccio classico per una ram dinamica. La dinamica è caratterizzata da certi tempi di accesso e di ciclo e quello che comanda realmente è quello di ciclo .Come miglioriamo le prestazioni della memoria? Si deve cercare di inviare un unico segnale RAS e più segnali CAS uno di seguito all'altro. Se noi immaginiamo la memoria come una matrice , viene inviato un indirizzo di riga e dopodichè mantenendolo costante vengono generati più indirizzi di colonna. Il vantaggio è il fatto che risparmiamo il tempo di indirizzamento della riga.

Questa modalità funziona solo se gli indirizzi generati sono consecutivi(ad esempio la memoria



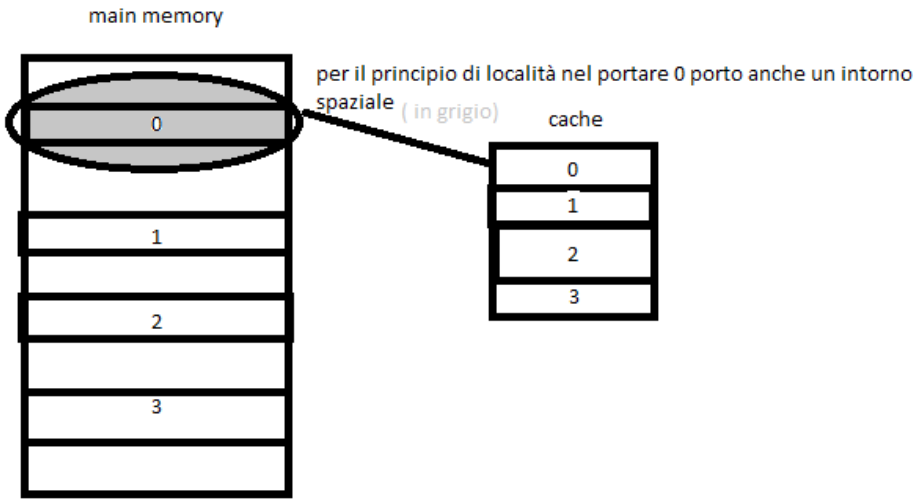
istruzioni dove le locazioni delle istruzioni sono consecutive). Questa è la modalità FAST PAGE MODE(FPM-RAM). In alcuni casi è sfruttabile in altri no. Questa tipologia ha una frequenza massima di 66 MHZ e la temporizzazione consecutiva è 6-3-3-3 ovvero che per accedere alla prima word ho bisogno di 6 cicli di clock e per le altre 3 word solo 3 cicli.
La EXTENDED DATA OUT(EDO-RAM) è una miglioria di quella precedente in quanto ottengo temporizzazione 5-2-2-2.Qui nel caco che il CAS quando viene disabilitato non viene disabilitata l'uscita e inoltre l' accesso avviene prima che venga concluso il precedente accesso.
Poi abbiamo la BURST EXTENDED DATA OUT(BEDO-RAM): riesce a generare autonomamente l'indirizzo di colonna. Questo rispetto a quello di primo non ha sostanziale miglioramento nei tempi. (slide 15) il passaggio importante si è avuto con l'introduzione delle memorie SINCRONE, quelle di prima sono ASINCRONE. Il processore va ad inizializzare la memoria dicendo che se ha bisogno di un certo dato dopo un certo tempo glielo si deve inviare. La memoria sa dopo quanti cicli di clock deve inviarlo. Tutto ciò viene fatto da una logica di controllo di un clock. I trasferimenti avvengono sul fronte del clock(o in fase di salita o in fase di discesa). Noi possiamo specificare un burst per il quale definire la dimensione. Dobbiamo definire quanti dati dobbiamo leggere o scrivere. Per quanto riguarda le temporizzazioni nel caso di un bus con frequenza a 100MHZ abbiamo bisogno di 6-1-1-1.Dopo la prima ad ogni fronte del clock abbiamo un nuovo dato. Ulteriore condizione di queste ram singole si ha con le DDR(memorie dinamiche singole usate nei calcolatori, oggi abbiamo le DDR3).
Qual'è la logica dietro le DDR? Nelle DDR anziché leggere o scrivere il dato ad ogni fronte di salita del clock(come nel caso della RAM sincrona classica)avviene sia nella salita e sia nella discesa, quindi raddoppio la velocità di trasferimento. Fisicamente io vado a raddoppiare il numero di bus interno alla memoria e quindi raddoppio il parallelismo. Quando mando l'indirizzo di riga anziché

leggerne una ne leggo contemporaneamente 2. Su un fronte mando il contenuto di un registro dall'altro il secondo registro. Quello che miglioro non è il tempo di accesso ma quello di trasferimento(il throughput). Questa cosa va bene quando noi abbiamo bisogno di più accessi consecutivi in memoria. (tipica situazione: abbiamo scheda video che deve trasferire immagini, tipicamente abbiamo grandi quantità di dati che si trovano in locazioni consecutive, questa modalità è quindi ideale!). Le prime DDR sono state utilizzate nelle schede video. Se vogliamo leggere invece un singolo dato non abbiamo nessun miglioramento rispetto alla RAM dinamica classica.



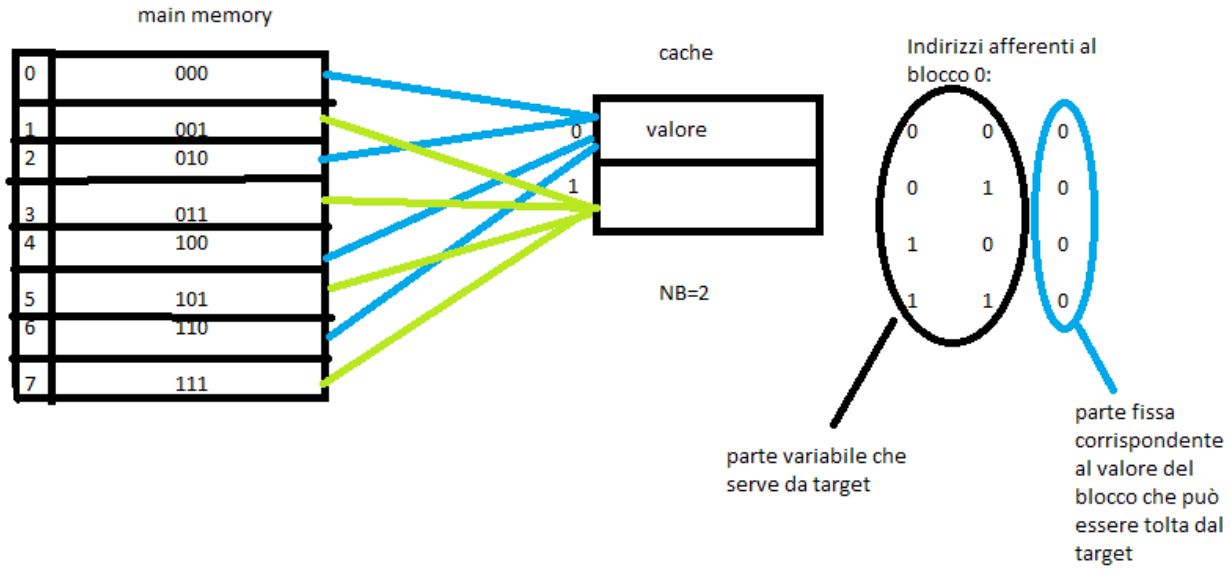
GERARCHIA DI MEMORIA

Con i sistemi visti precedentemente noi aumentiamo i tempi di trasferimento non i tempi di accesso. Queste latenze sono molto più alte rispetto a quelle tipiche di un processore. Per lavorare bene, per non essere considerato un collo di bottiglia rispetto all'intero sistema si dovrebbe avere che l'intera memoria dovrebbe essere fatta con RAM statica, ma ciò non è possibile per problemi di costo. Per questo motivo è stata creata una organizzazione di tipo gerarchico.



(slide 4) il processore inanzitutto cerca di accedere alla memoria cache(che è una ram statica). Se il dato di cui ha bisogno è presente all'interno della cache della ram statica allora l'accesso alla memoria viene completato, altrimenti quello che viene fatto è che il processore cerca di trovarlo nella main memory(ram dinamica) a questo punto preleva una o più word che trasferisce alla cache e successivamente il dato che serve alla CPU viene trasferito dalla cache al processore. Nel caso in cui i dati non sono presenti in RAM viene cercato questo dato ancora a livello più basso nell'hard disk. Noi ci concentreremo sostanzialmente a quello che avviene tra la memoria cache e quella principale. Il meccanismo di ciò che avviene tra ram e disco fisso è simile. La cache è una memoria piccola che dovrà contenere un sotto insieme della memoria principale. Grazie a questi due livelli è possibile creare l'illusione che i programmi accedano ad una memoria completamente statica perchè tipicamente nei programmi è presente LOCALITÀ SPAZIALE E TEMPORALE(se noi in un certo istante accediamo in una locazione i è molto probabile che fisicamente in un tempo t+epsilon io riferisca locazioni vicine a i, quello temporale dice che se accedo a i in un tempo t a t+epsilon probabilmente riuserò i questo perchè all'interno dei programmi ci sono dei cicli).Nel disegno 10 vediamo che come la cache contenga una sottocopia dello spazio della main memory. Abbiamo blocchi di word che sono presenti sia in main memory sia in cache. Quello che si cerca di fare e cercare di accesere inanzitutto in memoria cache come? se è presente una copia di memoria della main memory in cache viene letta dalla cache altrimenti si prende dalla main memory il dato che serve unito alle word che rispettano il principio di località e lo copio in cache. Questa è per la località spaziale. Se noi trasferiamo 0 dalla main alla cache in un tempo t+epsilon è probabile che non ci serva per la località spaziale un altro caricamento da main perchè ci servirà di nuovo quella.
Grazie a questo modo di procedere riusciamo a creare questa illusione di ram statica. Questa è tanto più verosimile quanto più frequente è il dato in cui un dato o un istruzione lo troviamo dentro la cache. Se noi nel 99% delle volte lo troviamo in cache sostanzialmente è come se vedessimo solo memoria statica. Quando troviamo un istruzione parliamo di HIT. se non lo troviamo in cache si parla di MISS. Invece il tempo richiesto per trasferire un blocco dalla main memory alla cache si indica come MISS PENALTY. Date queste tre grandezze noi possiamo calcolare quello che è il tempo medio di accesso alla memoria.
(slide 6) il tempo di accesso alla memoria Tamat= hit rate(frequenza degli hit)*Tts(tempo di accesso alla cache in caso di successo) + miss rate*(tp che sarebbe il miss penalty). Questa formula ci da il tempo medio di accesso. Hit rate=1- miss rate quindi formano un solo parametro. Quindi il tempo medio è dato da 3 dati.
Noi potremmo scegliere una politica che avvantaggia il miss penalty ma che mi svantaggia altri fattori. Noi dobbiamo trovare qualcosa che migliora il tempo medio non uno solo dei 3 parametri

perchè altrimenti se ne miglioro solo uno ho un peggioramento nelle prestazioni.
(slide 7)Ci sono quattro domande che ci dobbiamo porre:
(slide 8)dato un blocco di word dove lo dobbiamo andare a piazzare nella memoria? noi dobbiamo immaginare la cache e la main memory da un insieme di blocchi ognuno dei quali è formato da gruppi di word. Noi se trasferiamo un dato trasferiamo dei blocchi. Cambiando la dimensione dei blocchi le prestazioni del nostro sistema possono cambiare. Qui non ci interessa quanto è grande, e per il momento lo assumiamo un blocco formato da 4 word(che è un applicazione tipica). La cache immaginiamola composta da 128 blocchi e la ram da 1024 blocchi. Uno di questi 1024 blocchi dove lo piazzo dentro la cache? teoricamente lo potrei mettere in uno qualsiasi, ma se come politica di block replacement uso questa si usa una tecnica FULL ASSOCIATED(una qualsiasi delle posizioni è buona). Questa ha dei vantaggi dal punto di vista del miss rate ma crea problemi dal punto di vista dell'hit time.
Seconda soluzione opposta all'associativa completa è la DIRECT MAPPED. Qui esiste un unico posto all'interno della cache dove può essere piazzato. Questa posizione è ricavata mediante una formuletta index= j modulo NB(nb è il numero di moduli della cache) j è l'indirizzo di blocco della main memory e lo genera la CPU. in questo caso index= 130 modulo 128 ovvero 2. Se noi prendiamo j=2 allora index= 2 modulo 128 che è sempre 2. Se noi andiamo a prendere j=258 l'index corrispondente è 258 modulo 128 ovvero 2. se prendiamo j=514 index=2. Come vedete tutti i moduli che all'interno della main distano 128(quindi NB)verranno mappati tutti nello stesso blocco della cache. Con questa politica più blocchi sono mappati nello stesso blocco dellea cache. Dal punto di vista del tirare fuori il numero di index, il procedimento è molto semplice, basta andare a prendere dall'indirizzo generato dalla CPU un certo numero di bit significativi. Il numero di bit corrisponde al log in base due di NB. Nel caso di prima avevamo 128 blocchi e quindi per indirizzarli abbiamo bisogno di quanto? 7 bit. Con 7 bit meno significativi dell'indirizzo prodotto dalla CPU indirizziamo i blocchi della cache.
(slide 9) esiste un associazione tra i due approcci. Si chiama N-WAY SET ASSOCIATIVE. In pratica la memoria cache viene divisa in insiemi ciascuno dei quali contiene un certo numero di blocchi. In questo caso andiamo a piazzare un blocco della cache in uno dei set usando una formula simile. Index=j modulo NS(numero di set). All'interno di set noi siamo liberi di andare a salvare il blocco dove vogliamo. Un set può essera fatto ad esempio da 4 blocchi. Volendo il primo blocco lo possiamo pensare come un estremizzazione di questo secondo metodo.
(slide 10) qui vedete un disegno che fa vedere quanto abbiamo detto a parole .Immaginiamo di avere la main memory e una cache di 7 blocchi. Nel caso Full associative può essere mappata in qualunque blocco può essere associata sono al numero 4, invece nel set associative posso metterlo nel set 0 che è formato da due blocchi, quindi in uno qualsiasi di questi 2. Vediamo di affrontare un'altro problema come facciamo capire se c'è stato un hit o un miss ed è il Q2.
(slide 11)Q2: per capirlo facciamo un esempio semplice(disegno 11) abbiamo una main di 8 blocchi



e una cache di 2 blocchi. In questo caso NB=2. Utilizzando la direct mapped, osserviamo che il blocco 0 della main verrà piazzato nel blocco 0 della cache. E così tutti i blocchi pari, mentre i dispari in uno. Poichè 4 blocchi della mani vanno nel blocco della cache come faccio a capire tra i 4 blocchi che potenzialmente possiamo avere qual'è quello giusto che stiamo cercando?Noi abbiamo un VALORE ma come faccio a capire se è quello che sto cercando? se io mi limito a copiare il valore io non sono in grado di capirlo. Per capirlo dovrei associare assieme al valore anche un ID che univocamente mi identifichi quel blocco e quale può essere questo ID? è l'indirizzo della main associato a quel valore, ovvero se io ho copiato il blocco corrispondente a 000 oltre a memorizzare il valore vero e proprio vado a mettere insieme al valore la configurazione 000 a cui corrisponde quel valore in cache. Se invece quel valore non è di 000 ma di 010 assieme a quel valore metto 010. Ma se osserviamo in tutti quegli indirizzi c'è lo zero finale che è costante!!quindi in realtà invece di 3 bit ne posso memorizzare solo 2. Se notiamo la parte costante corrisponde all'indice dei blocchi in cache. Quindi anzichè memorizzare l'intero indirizzo memorizzo la parte variabile dell'indirizzo la TAG (nel nostro caso solo 2 bit). Come otteniamo questo tag a partire dall'indirizzo generato dalla CPU? l'indirizzo è fatto da TAG,INDEX e OFFSET. Quest'ultimo mi dice qual'è la word che ci serve delle 4 nel nostro esempio. L'index mi permette nella direct o nell'associative di individuare in modo diretto il blocco o il set. Poi abbiamo la parte restante dell'indirizzo che è la parte variabile, questa la memorizziamo assieme al valore. Questo è il procedimento per capire se c'è hit o miss.
(slide 12) se quello è l'indirizzo dall'index accediamo direttamente al blocco della cache, nella tabella a destra vi è il dato e in quella sinistra il tag. Il tag viene confrontato nel" ?" con il tag della main memory, se coincidono allora il blocco presente in cache è quello che stiamo cercando e quindi abbiamo un HIT.
(slide 14)nel caso ASSOCIATIVO qua abbiamo la parte di memoria contenenete i valori nella tabella data e i tag nella tabella a fianco. Notiamo la presenza di un multiplexer che non è presente nello schema prima, questo riceve un blocco da una parte e uno da sotto, poi abbiamo una linea per l'hit di sopra e uno per quello di sotto, ovviamente solo uno dei due può avere un hit non possono essere contemporanei.Se uno dei 2 è attivo in uscita ci da la word corrispondente all'hit attivo. La presenze del MUX ha un effetto deleterio nel tempo di accesso della memoria. Perchè oltre al tempo di accesso devo considerare il tempo di latenza. Se noi anzichè avere una cache associativa a 2 vie c'è l ho a 4 vie aumenta la latenza e così via.
(slide 15)Q3: quale blocco devo sostituire nel caso di miss? questo problema non si pone nella direct in cui esiste un unico posto dove lo posso mettere.Nel caso dell'associativa avendo un set di più blocchi dobbiamo scegliere. Se entrambi i blocchi sono scritti quale devo sovrascrivere? posso fare o RANDOM (scelgo a caso) o LAST RECENTLY USED(scarto quella che da più tempo non uso). Attraverso un analisi quantitativa vedo che per un numero di vie più basso con dimensioni piccole di cache la seconda politica ha prestazioni migliori, però aumentando la dimensione della cache le prestazioni delle due politiche sono abbastanza simili.
(slide 16)Q4: cosa succede in caso di scrittura. La memoria cache la possiamo immaginare come una copia dei blocchi in main memory. Ora supponiamo di fare un operazione di scrittura. la possiamo fare soltanto lavorando sulla copia presente in cache non scrivendo in main memory. In questo caso non sempre la copia in cache sarà uguale a quella presente in main. Cosa possiamo fare? nel caso in cui decidiamo di scrivere soltanto in cache, quando dovrà essere scartato se il blocco da buttare è stato modificato dobbiamo aggiornare il contenuto della main con la cache. Dopo ciò posso modificare. Questo approccio è chiamato WRITE BACK. Il WRITE THROUGH è che in caso di scrittura operiamo contemporaneamente sia in cache sia in main memory. Questo conviene con poche scritture ma se ne ho tante non converrebbe. per eliminare questo tipo di problema insieme a WT , si utilizza il write buffer(tra cache e main) che nel caso di scrittura io scrivo contemporaneamente nel cache nel Write buffer, poi successivamente va a scrivere in main memory. Questo funziona perchè normalmente nei programmi non abbiamo 10 scritture consecutive ma poche una dopo l'altra. Se il WB è grande da poterne conservare un certo numero consecutive, io le scrivo li e poi successivamente mentre io non faccio scritture lui li scriverà in main memory. Quindi il processore non si blocca in attesa che venga fatta la scrittura.

IL SISTEMA DI I/O

(slide 1)Tipicamente un sistema di elaborazione è caratterizzato da un insieme di periferiche che comunicano non direttamente alla CPU ma attreverso una interfaccia. La cpu vede le interfaccie che servono per realizzare sia un adattamento logico ed è caratterizzato da un insieme di registri attraverso il quale è possibile attuare il trasferimenti da dati dalla cpu alla periferica e viceversa e possiamo comandare lettura e scrittura.
(slide 2) Tipicamente una periferica è caratterizzata da 3 registri. Il DREG è il registro contenente il dato da trasferire. Il dato sarà scritto dalla periferiche e letto dalla CPU, nel caso contrario è la CPU a scrivere in questo registro. L'SREG
CREG è un registro grazie al quale settato dalla CPU vengono richieste certe operazioni alla periferica. Ciascuna di queste interfaccie associate alle periferiche è caratterizzata da un certo indirizzo. La CPU dovrà esplicitamente dire l'indirizzo della periferica con qui vuole colloquiare.
(slide 3) abbiamo fondamentalmente due modalità con cui noi possiamo andare ad indirizzare le periferiche. Il primo approccio è il memory mapped I/O. Se noi vogliamo andare a leggere o scrivere un dato su una periferica utilizzeremo un istruzione identica a quella per accedere ad una zona di memoria (come la load per esempio). Come facciamo a capire se abbiamo bisogno di un dato relativo ad una periferica? c'è lo abbiamo perchè l'indirizzo non è relativo ad una zona di memoria ma relativo ad una periferica.Immaginiamo lo stato di indirizzamento della CPU in questa forma.Se l'indirizzo prodotto dall'istruzione fa riferimento alla parte I/O farà riferimento ad una periferichа altrimenti alla memoria. In I/O avremo mappate le interfaccie relative alle periferiche.Per esempio 1000 indica in modo univoco la periferica 1, 1004 la 2... Una soluzione alternativa è l'I/O mapped. In questo caso noi utilizziamo delle istruzioni specifiche per indirizzare le periferiche. Ad esempio IN INDIRIZZO E OUT INDIRIZZO. Dobbiamo considerare gli spazi di indirizzamenteo disgiunti. Noi potremmo avere un indirizzo 0 per la memoria e uno 0 per la memoria. 0 IN significa indirizzo della periferica mappata a 0, invece load 0 fa riferimento alla lettura di una locazione di memoria di indirizzo 0.Out è per scrivere un valore su una certa periferica. Nel caso di un processore MIBS l'approccio utilizzato è il primo.
(slide 4) andiamo a vedere come vengono gestite le periferiche ovvero come viene gestita la comunicazione tra CPU e periferica. Abbiamo 3 tecniche.
Oggi vedremo la prima che è il POLLING.
(slide 5) POLLING in questo caso la CPU va a controllare ad uno ad uno il registro di stato dell'interfaccia di ogni periferica. Se una periferica ha bisogno di comunicare un dato allora la CPU analizzando il bit di stato di quella periferica vede che la periferica è già pronta. VEDI abbiamo una sequenza di istruzioni che va a valutare ad una ad una. Se è pronta per l'operazione di I/O allora intraprende quello che deve fare la periferica. Qual'è il problema di questo tipo di soluzione? dal punto di vista hardware è incredibilmente semplice perchè si tratta di eseguire soluzioni di controllo che mi fanno perdere tempo di CPU. QUindi una parte elevata del tempo di CPU viene spesa per fare un controllo del registro di stato. In questo intervallo non può eseguire la CPU. Poichè il controllo del registro di stato tramite il polling viene fatto attraverso una regione sequenziale le periferiche che vengono controllate dopo sostanzialmente hanno una priorità più bassa rispetto a delle periferiche che hanno priorità nel polling. E' chiaro che utilizzando una tecnica di questo tipo una periferica che viene controllata alla fine sarà servita dopo molto tempok, quindi non è una politica equa.
(slide costo polling) vediamo che una politica di controllo sulle periferiche è buona o no. NEl primo caso usiamo il polling per gestire il mouse, nel secondo per il floppy e poi nel terzo per l'hard disk. Per quanto riguarda il mouse bisogna valutare i movimenti x/y del mouse 30 volte al secondo, immaginando che un ciclo di polling intero conti 400 giri di clock , vogliamo sapere quant'è la percentuale di CPU spesa per il polling. Per poter fare un polling tale da testare 30 volte al secondo il mouse servono 12000 cicli. e la percentuale è di 0,002% . L'uso del polling per gestire il mouse va bene. Questa è quindi una tecnica utilizzata per dinamiche come quelle del mouse. Vediamo cosa accade nel caso del floppy. Questo è in grado di trasferire 2 byte con velocità di 50 KB al secondo.

il polling si calcola per ogni coppia di byte trasferiti.I cicli di polling al secondo sono 10milioni . Considerando una frequenza di 500 mhz la percentuale è del 2% sempre bassa comunque. Se invece consideriamo l'hard disk ipotizzando 8 mb al secondo di velocità. I cicli di polling qui sono 200 milioni di cicli al secondo che portano ad una percentuale del 40% che è inaccettabile.

LEZIONE 09-06-2011

una soluzione che può ridurre i problemi del polling è il sistema di interruzione. Così, non è più necessario che sia la cpu a interrogare le periferiche, perchè, invece, saranno le periferiche ad avvertire la cpu quando sono pronte a fare un trasferimento, mandando una richiesta di interruzione(IRQ). Non appena la periferica fa un IRQ, la cpu termina l'esecuzuione dell'istruzione corrente e manda in esecuzione una routine di servizio per quella richiesta, cioè una sequenza di istruzioni che permette il trasferimento richiesto dalla periferica, quindi si evita che la cpu spenda istruzioni non utili per cercare di capire se la periferica ha bisogno di eseguire un trasferimento. L'IRQ è asincrona rispetto alle esecuzione delle istruzioni quindi può arrivare in qualsiasi momento. Nel caso in cui il nostro processore sia sequenziale, l'IRQ potrà essere valutata solo alla fine dell'EX dell'istruzione corrente. Invece, se il processore è pipeline, significa che vi sono più istruzioni all'interno della pipe, quindi ci si pone il problema di quando valutare l'IRQ.

Esempio:

IF	ID	EX	MM	WB		
	IF	ID	EX	MM	WB	
		IF	ID	EX	MM	WB
		IRQ=1				

Supponiamo che nel ciclo di clock indicato sopra (terza colonna) viene inviata una IRQ. La prima istruzione è in fase di EX, la seconda in fase di ID e la terza in fase di IF.
Quali istruzioni dobbiamo considerare concluse per avviare la routine di servizio?
Esistono diverse soluzioni...
Ad esempio, quando arriva l'IRQ, potrei completare tutte le istruzioni nella pipe e inibire i fetch delle nuove istruzioni che vogliono entrare nella pipe. Così facendo, dovrei inserire una serie di stalli nella pipe per tutti i cicli di clock che vanno dal completamento delle istruzioni che si trovano già dentro la pipe e l'esecuzione della routine di servizio. Una volta completata la routine di servizio, l'istruzione che verrà eseguita sarà quella che sarebbe andata in esecuzione se non vi fosse stata l'interrupt (questa viene chiamata interruzione precisa).
Altre soluzioni (interruzioni imprecise) prevedono che la routine venga eseguita immediatamente dopo l'IRQ e le istruzioni dentro la pipe vengano buttate fuori dalla pipe e salvate per essere eseguite subito dopo la routine per garantire la correttezza del programma corrente.

Funzioni del sistema di interruzione
Il sistema di interruzione deve:
- garantire che non vi siano interferenze sul programma in corso;
- riconoscere il dispositivo da cui parte l'IRQ;
- gestire le priorità nel caso in cui più dispositivi inviino l'IRQ nello stesso ciclo di clock. Una periferica con priorità più alta può interrompere l'esezione della routine di una periferica con priorità più bassa. Il viceversa non è possibile.

Salvataggio del contesto
Riassumento, prima di mandare in esecuzione la routine di servizio, il contesto del programma interrotto verrà salvato in un'area di memoria. Il PC assume il valore dell'indirizzo della prima istruzione della routine di servizio, la quale viene eseguita. Dopo l'esecuzione il programma riprende dall'istruzione successiva a quella interrotta.
Il PC e il registro di stato vengono salvati automaticamente via hardware.
Se la routine andasse a modificare il contenuto di una locazione di memoria usata dal programma interrotto, dovremmo considerare anche tali aree di memoria come parti del contesto e quindi salvarle. I contenuti di tali aree di memoria e i registri di uso generale vengono salvati via software nello stack (push), cosa che tipicamente accade grazie al preambolo della routine di servizio, prima che la routine stessa li vada a modificare.
Il salvataggio del contesto non può essere mai interrotto, ad eccezione del verificarsi di eventi catastrofici (es, viene meno la corrente di alimentazioni e si preferisce annullare il salvataggio del contesto per salvare alcuni dati fondamentali). Per evitare che il salvataggio venga interrotto c'è il flag IE che è settato a 1 se il programma può essere interrotto. Quando arriva l'IRQ e si avvia il salvataggio, IE=0 e quindi tutti gli interrupt vengono ignorati. A salvataggio concluso, IE viene portato di nuovo a 1 via hardware oppure via software immettendo una certa istruzione.

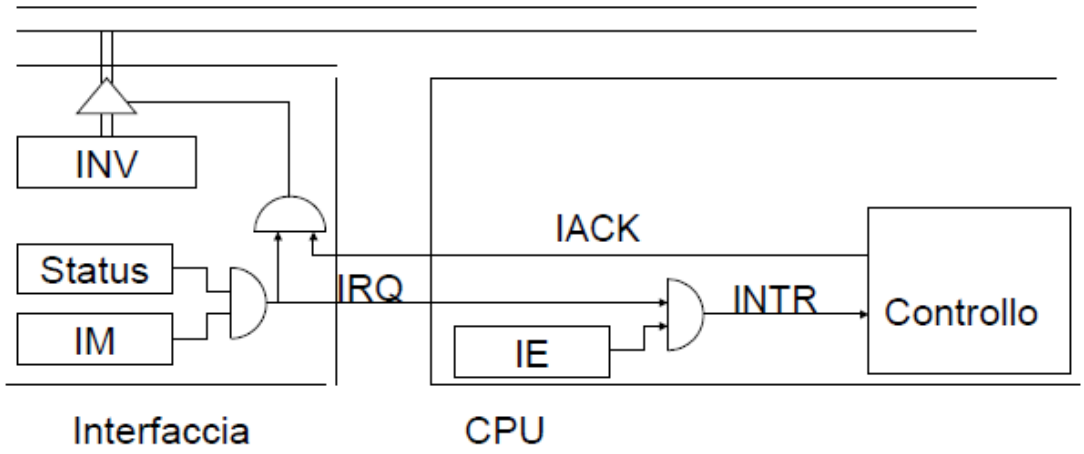
Ripristino del contesto
Dopo l'esecuzione della routine di servizio:
- vengono ripristinati via software i valori dei registri salvati nello stack (pop).
- vengono ripristinati via hardware il PC e il registro di stato.
Durante il ripristino del contesto IE=0. Dopo il ripristino IE=1.

Interruzioni vs polling
Qual è la percentuale di tempo spesa nei trasferimenti col polling e con l'interrupt?
hp: i trasferimenti vengono fatti su un hardisk di 8MB/sec, per ogni trasferimento vengono trasferiti 16B. Ogni trasferimento richiede 500 cicli di clock con una frequenza pari a 500MHz. L'hardisk è attivo solo il 5% del tempo.
Col polling, la cpu monitora costantemente l'hardisk, anche se non è attivo. Si spendono 250 milioni di clock al secondo. Il 50% del tempo della cpu viene usato per gestire i trasferimenti.
Con il sistema di interrupt, si hanno 500K interruzioni al secondo, che in linea teorica, sarebbe un valore 500 volte inferiore rispetto a quello del polling.

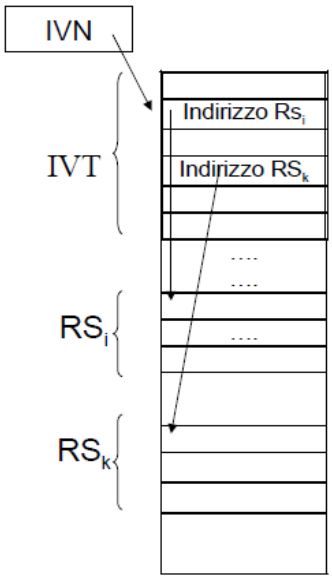
Interruzioni non mascherabili
Nella cpu abbiamo un ingresso attraverso cui passano le interruzioni non mascherabili, che sono dei segnali che non possono essere ignorati in quanto vengono usati solo in situazioni di emergenza (ad esempio, viene usata per salvare dati nel caso vi sia un calo di tensione).

Riconoscimento del dispositivo di I/O
Come fa la CPU a capire quale periferica ha fatto l'IRQ?
Approccio software: si effettua un polling delle periferiche e non appena si trova la periferica il cui stato è a 1, viene bloccato il programma corrente e si esegue la routine di servizio associata a quella periferica. E' una soluzione semplice dal punto di vista hardware, ma ha alcuni difetti:
- se la periferica che ha fatto al richiesta è alla fine della scansione, la risposta non sarà immediata, specie se si ha un gran numero di periferiche;

- l'ordine in cui viene fatta la scansione definisce la priorità delle periferiche, ossia le ultime ad essere scansionate hanno priorità più bassa. Questo si può risolvere aggiungendo un meccanismo di round robbing per cui si cambia dinamicamente l'ordine di scansione delle periferiche.
Approccio hardware: Interruzione vettorizzata
Ogni periferica ha un numero identificativo univoco(interrupt vector number, INV).



Supponendo che IM=1 (cioè l'interrupt è mascherabile) e IE=1, se una periferica vuole comunicare con la cpu (status=1), viene inoltrata una IRQ all'unità di controllo della CPU e, a questo punto, la CPU risponde attivando l'interrupt acnowledge (IACK=1). L'IACK insieme all'IRQ fanno in modo che l'uscita dell'AND in cui sono entrambe in ingresso sia 1 e quindi abilita il buffer tri-state che ha come ingresso il vettore INV che verrà mandato sul bus dati che a sua volta verrà letto dalla CPU. Nello specifico, l'INV è un vettore che rappresenta l'indice della interrupt vector table (IVT) che corrisponde alla routine di servizio. Detto in maniera più zaurda, l'INV è un vettore di puntatori alle locazioni di memoria della IVT che contengono i puntatori alle routine di servizio. La IVT di solito è nella parte iniziale della memoria, cosicchè possiamo utilizzare pochi bit per identificare tali locazioni.



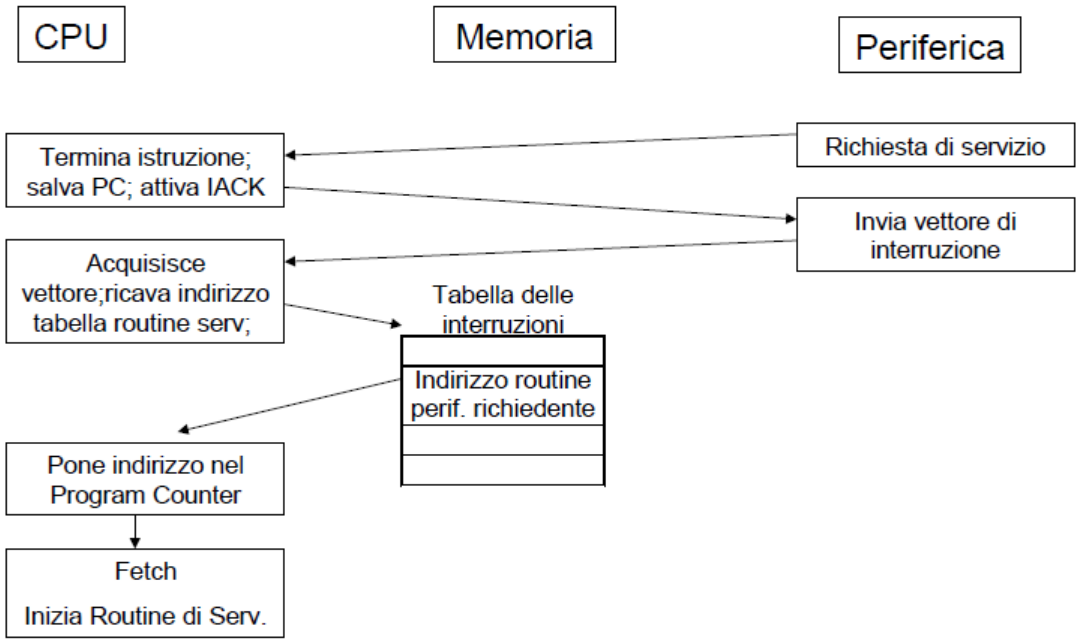
Supponiamo che la tabella sopra comprenda i primi 1024 byte e immaginiamo che ogni locazione sia una word di 32 bit. In totale avremo 256 word da 4byte che potrebbero essere 256 routine di

servizio che ci potrebbero permettere di controllare 256 periferiche. Per utilizzare ogni word, dei 32 bit a disposizione ce ne basteranno 8.

000.....000 | 00000001 |00
22 bit 8 bit 2bit

A partire dagli 8 bit (che ci permettono le 1024 combinazioni che ci servono) ci basta aggiungere due zeri per richiamare l'indirizzo del primo byte di ognuna delle 256 word (ricordiamo che ogni word è di 4 byte, quindi ogni primo byte di ogni word sarà multiplo di 4 che in binario si rappresenta come 100... e se sono tutti multipli di 4 i due zeri meno significativi ci saranno sempre). Gli altri 22 bit sono tutti a zero. Seguendo questo criterio, la stessa tabella si può ricostruire anche in altre parti della memoria eseguendo opportune modifiche.

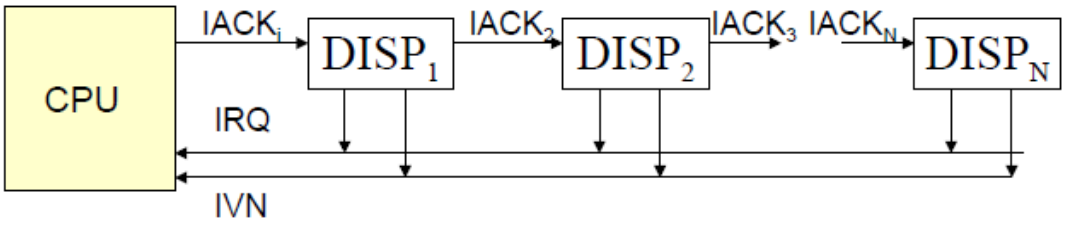
Riconoscimento mediante vettore delle interruzioni



Conflitti

Più interruzioni possono entrare in conflitto. Le cause possono essere:
- richieste contemporanee;
- richieste quando è in esecuzione la routine di servizio di una interruzione.

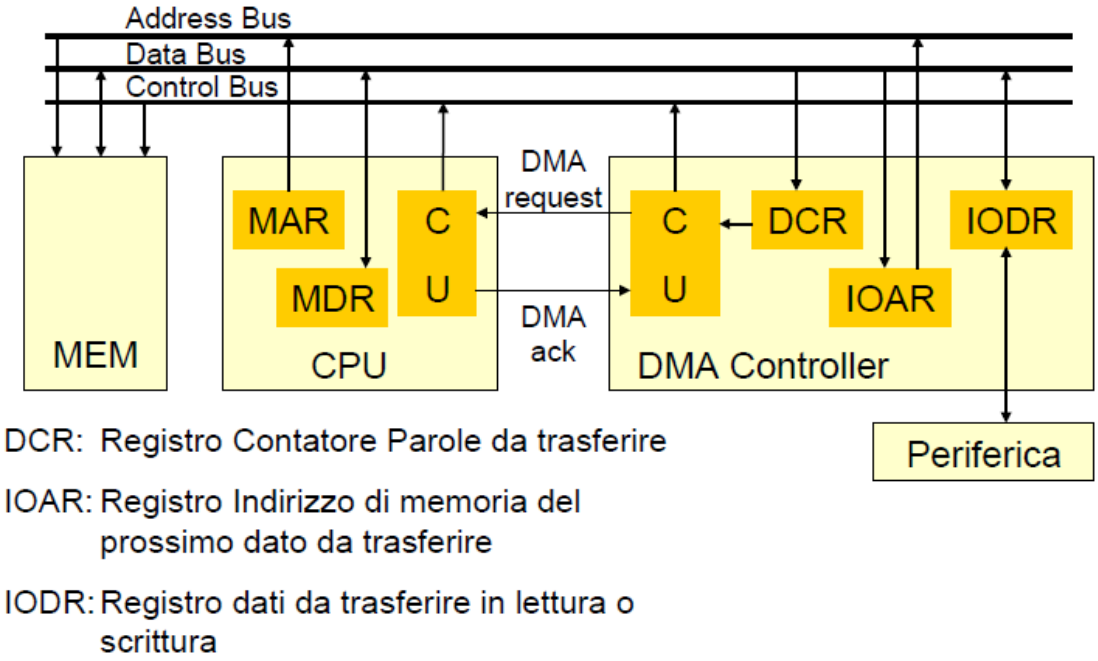
Un modo per risolvere il problema delle richieste contemporanee è usare il meccanismo di interruzioni vettorizzate con Daisy Chain.



Supponiamo che uno dei disp fa una richiesta di interruzione. La CPU riceve l'IRQ e manda l'IACK. L'IACK arriva a DISP1. Se è stato DISP1 a inviare l'IRQ, in uscita manderà IACK2=0 e quindi il dispositivo 2 e tutti gli altri vedranno sempre uno IACK=0, quindi anche se gli altri DISP inviassero una IRQ non avrebbero mai l'IACK desiderato e non si eseguirà il loro interrupt finchè l'interrupt di DISP1 non avrà finito la sua esecuzione. Chiaramente, la priorità è statica e dipende dalla posizione dei dispositi: più il dispositivo è lontano, minore sarà la sua priorità.
Un'alternativa simile al Daisy chain, potrebbe comporsi di una cpu che comunica con un modulo che a sua volta è collegato a tutti i dispositivi in maniera parallela. In questo modo, sarebbe il software del modulo a stabilire la priorità.

Interruzione della routine di servizio
Approccio software: nel preambolo della routine di servizio, oltre a salvare il contesto definiamo l'insieme di periferiche che possono interrompere la routine di servizio in corso di esecuzione settando l'IM di tali periferiche. Questo rallenta l'esecuzione della routine di servizio.
Approccio hardware: ad ogni periferica viene associato un livello di priorità (più periferiche possono anche avere lo stesso livello di priorità). Il processore conserva in un apposito registro il valore corrente della priorità. Quando è in esecuzione un programma normale il livello di priorità del registro è 0. Non appena viene mandata in esecuzione la routine, il registro assume il valore del livello di priorità della routine. Se mentre è in esecuzione tale routine, arriva una richiesta con priorità inferiore, essa viene ignorata. Se, invece, la priorità della nuova richiesta è più alta rispetto a quella corrente, viene interrotta la routine in corso e viene il registro assume il nuovo valore di priorità. Il contesto e la prioritò della routine interrotta verranno conservati in modo tale che non appena viene conclusa l'esecuzione della routine interrompente, verrà subito eseguita la routine interrotta.

DMA
Grazie all'interruzione alla memoria si risparmia in termini di tempo di gestione del trasferimento. In alcuni casi, il tempo speso per i trasferimenti può comunque essere eccessivo oppure l'intervento della cpu sui trasferimenti potrebbe non consentire velocità elevate (ad esempio, nel caso in cui servano tante istruzioni per trasferimenti di pochi dati).
Per risolvere questo tipo di problema, è stato inventato il DMA controller che serve a bypassare la cpu. Quando i dati sono pronti al trasferimento dalla periferica, il DMA controller notifica alla cpu che è pronta al trasferimento. La cpu rilascia il bus per il DMA controller che realizza il trasferimento prendendo i dati dal registro della periferica e li trasferisce in memoria molto velocemente. Al termine del trasferimento il bus viene restituito alla cpu.
Il trasferimento è chiaramente più veloce.



I registri del DMA controller sono descritti nella figura sopra. IOAR e DCR vengono inizializzati dalla CPU. A seguito di ogni trasferimento, DCR e IOAR vengono decrementati. Quando DCR assume il valore zero vuol dire che il trasferimento è stato completato

Il DMA controller invia una richiesta di trasferimento mediante il DMA request. La cpu, dopo un certo tempo, rilascia i 3 bus e attiva il DMA ack. Il DMA riceve l'ack e prende possesso dei bus e comincia il trasferimento, ovvero manda sul bus indirizzi il contenuto del registro IOAR, manda sul bus dati il contenuto di IODR; manda sul bus di controllo i segnali che abilitano la scrittura in memoria.

Il valore di IODR cambia, e al termine della scrittura DCR e IOAR vengono decrementati per puntare alla locazione di memoria successiva. I bus vengono riempiti di nuovo con i nuovi valori dei registri e i contatori vengono decrementati. Questo si ripete finchè il DCR non è uguale a zero e quindi il trasferimento è completato.

Quindi il DMA request e il DMA ack vengono messi a zero e il DMA controller restituisce il controllo dei bus alla cpu.

Modalità di trasferimento

Il trasferimento dei dati in DMA può avvenire in diversi modi:

- trasferimento a burst: Si tratta di trasferimenti grandi blocchi di memoria, e quindi il DMA prende possesso dei bus per tanto tempo lasciando la cpu inattiva. Viene utilizzato quando ad esempio dobbiamo trasferire blocchi di dati dall'hardisk alla memoria.
- cycle stealing: trasferimento di piccole quantità di dati. Inizio il trasferimento, facendo prendere il controllo dei bus al DMA, ma dopo un certo tempo il DMA restituisce il controllo dei bus alla cpu, la quale a sua volta dopo un tempo prestabilito restituisce i bus al DMA... e così via. Questo alternarsi evita che l'uno o l'altro rimangano inattivi troppo a lungo.