Calcolatori Elettronici L-A

ISA DLX: Implementazione Tramite Struttura
"Pipelined"

Principio del Pipelining (1)

Il pipelining è oggi la principale tecnica di base impiegata per rendere "veloce" una CPU

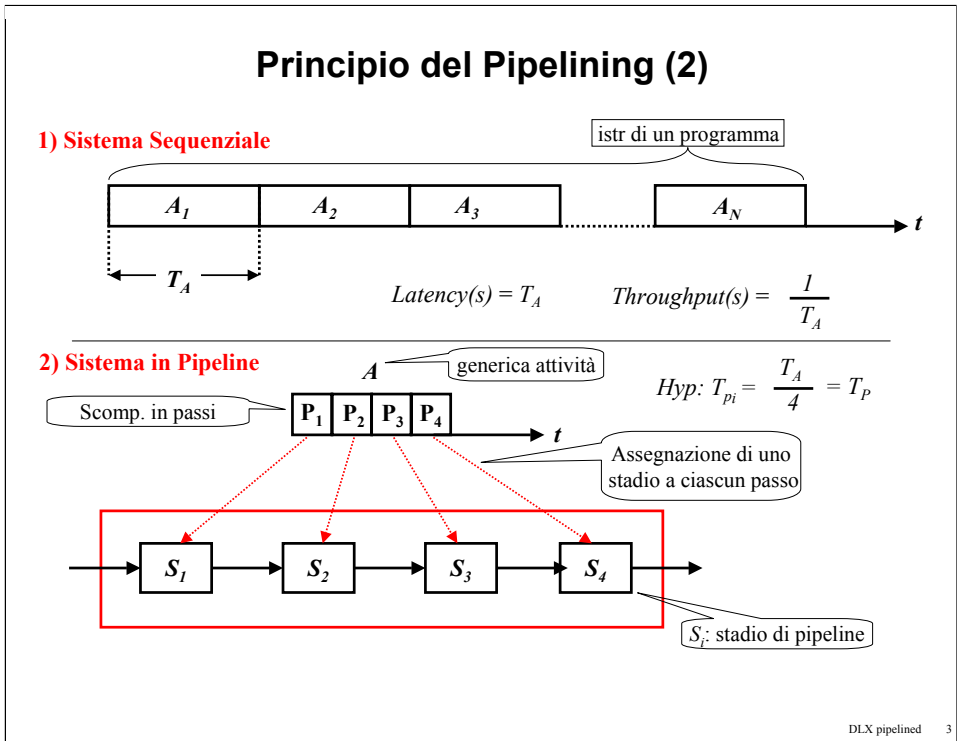
L'idea alla base del pipeling è generale, e trova applicazione in molteplici settori dell'industria (linee di produzione, oleodotti …)

Un sistema, *S*, deve eseguire *N* volte un'attività *A*:

Def.: *Latency*: (o latenza [sec]) T_A tempo che intercorre fra l'inizio ed il completamento dell'attività *A*.

Def.: *Throughput* : (produttività [Hz]) frequenza con cui vengono completate le attività ($\frac{1}{T_A}$)

DLX pipelined 2



P = PIPELINED S=SEQUENZIALE

1) Un programma è fatto di molte istruzioni diverse tra loro. In questa fase, per semplificare il formalismo ed i calcoli si assume che le istruzioni richiedano tutte gli stessi passi per essere eseguite e durino tutte lo stesso tempo (abbiano la stessa latenza).

Per enfatizzare il concetto di considerare il carico delle istruzioni uguale per tutte, chiamiamo la generica istruzione "attività" e immaginiamo che questa sia ripetuta più volte nel programma (come se si trattasse di un loop).

La latenza e il throughput di una esecuzione flat si calcolano come mostrato...

2) nel sistema in pipeline una attività è scomposta in PASSI (P1,..,P4) ed a ciascuno di essi è dedicato uno stadio della pipeline (S1,..,S4) specifico per l'esecuzione del passo.

L'intervallo di tempo Tp [sec], detto "cycle time" della pipeline, costituisce la durata dell'intervallo elementare di funzionamento e determina il ritmo con cui le attività attraversano la pipeline: tutti gli stadi sono perfettamente sincronizzati in modo che allo scadere di ogni intervallo di funzionamento una attività viene completata, ed abbandona l'ultimo stadio della pipeline, una nuova attività viene inserita nel primo stadio della pipeline e le altre attività presenti avanzano di uno stadio.

Intuitivamente Tp può essere considerato pari ad un ciclo di clock ma può durare anche più di un Tck, l'importante è:

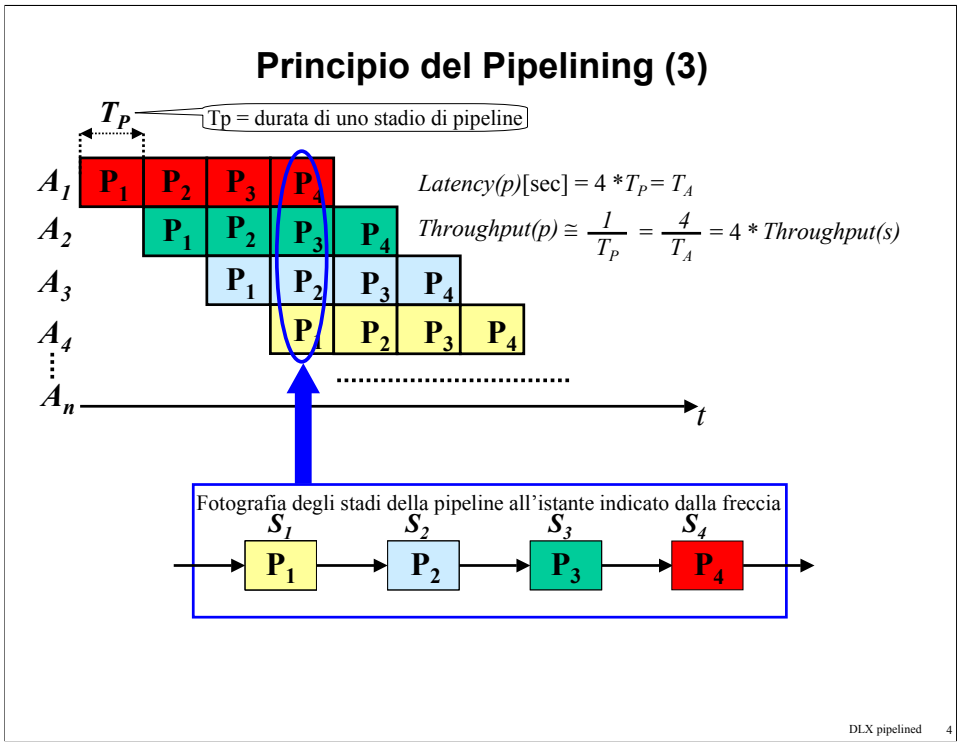
• che Tp sia uguale per tutti gli stadi

• che Tp sia pari al tempo richiesto dallo stadio più lento

• che non risulti conveniente scomporre lo stadio più lento in sottostadi la cui durata di elaborazione coincida con quella degli stadi più veloci ...come si può vedere dall'ipotesi nel lucido

Nella pipeline un'attività è scomposta in passi ed elaborata un passo per ciascun Tp. Ad un dato Tp si elabora il primo passo, al Tp successivo il secondo passo...

Ad ogni Tp una certa attività Ai può essere presente ed elaborata in un solo passo (o stadio)



Nello schema un'unica pipeline è rappresentata su più righe perché ad ogni riga corrisponde la successione temporale per una certa attività. Righe diverse rappresentano l'entrata in pipeline di attività diverse (colori diversi) ma si noterà che per un certo istante (ascissa) gli stadi impegnati in una colonna sono tutti diversi. Se vogliamo avere una idea del contenuto degli stadi della pipeline in ogni istante dobbiamo guardare le colonne come nel caso della "fotografia" in cui si nota che ogni stadio "Si" è impegnato ad eseguire il passo che gli compete Pi per una diversa attività (colori). Mano a mano che ci si sposta da sinistra a destra lungo una riga (+ tempo), si nota che viene eseguito Passo diverso per la attività Ai di quella riga.

Ciò che cambia rispetto alla cosiddetta esecuzione sequenziale è che si cerca di rendere gli stadi sempre occupati, perciò più istruzioni sono elaborate contemporaneamente ma ciascuna in uno stadio diverso.

Principio del Pipelining (4)

Il pipelining *non riduce il tempo necessario al completamento di una singola attività:*

$$Latency(p)\ [\text{sec}] = Latency(s)$$

Il pipelining *incrementa il Throughput*, moltiplicandolo per un fattore pari al numero degli stadi (K):

$$Throughput(p)\ [1/\text{sec}] = K * Throughput(s)$$

Ciò porta ad una riduzione dello stesso fattore *del tempo totale di esecuzione* di una sequenza di N attività T_N = [sec]:

$$T_N = \frac{N}{Throughput} \rightarrow T_N(s) = \frac{N}{Throughput(s)}\ ,\quad T_N(p) = \frac{N}{Throughput(p)}$$

$$\Rightarrow Speedup\ (p) = \frac{T_N(s)}{T_N(p)} = \frac{Throughput(p)}{Throughput(s)} = K$$

DLX pipelined 5

Facciamo un esempio con 4 stadi.

In generale in un sistema dotato di pipeline:

•l'attività che deve essere eseguita viene decomposta in una sequenza di *K passi elementari* ed il sistema strutturato come una cascata di *K unità funzionali*, dette *stadi*, ciascuna dedicata all'esecuzione di un passo elementare.

•gli stadi sono sincronizzati: è possibile iniziare una nuova attività alla scadenza di ogni intervallo elementare di funzionamento, ottenendo a regime l'*esecuzione simultanea di K attività*.

Quindi il pipelining, pur non modificando il tempo di esecuzione di ciascuna delle N attività eseguite dal sistema, rende il sistema K volte più veloce. La ragione è che il pipelining è una *forma di parallelismo*: ci sono K attività eseguite contemporaneamente (ma la pipeline è unica e c'è un solo stadio per ogni attività; i sistemi con più pipeline si chiamano superscalari, es.: intel dal primo Pentium in su).

Bilanciamento di una Pipeline

• Caso Ideale

$T_P = T_{pi} = \frac{T_A}{K}$ ⇒ Pipeline perfettamente bilanciata ⇒ $Speedup = K$

• Caso Reale

$T_P = max\ (T_{P1}, T_{P2} ..... T_{PK})$ ⇒ Bilanciatamento imperfetto ⇒ $Speedup < K$

• Un Esempio

$T_A = 20\ t$ (t : unità di tempo, es.: cicli di clock)

$T_{P1} = 5\ t$, $T_{P2} = 5\ t$, $T_{P3} = 6\ t$, $T_{P4} = 4\ t$ ⇒ $T_P = 6\ t$ (è il più lento)

⇒ $Speedup\ (p) = \frac{T_A}{T_P} = \frac{20\ t}{6\ t} \cong 3.33\ (< 4)$

DLX pipelined 6

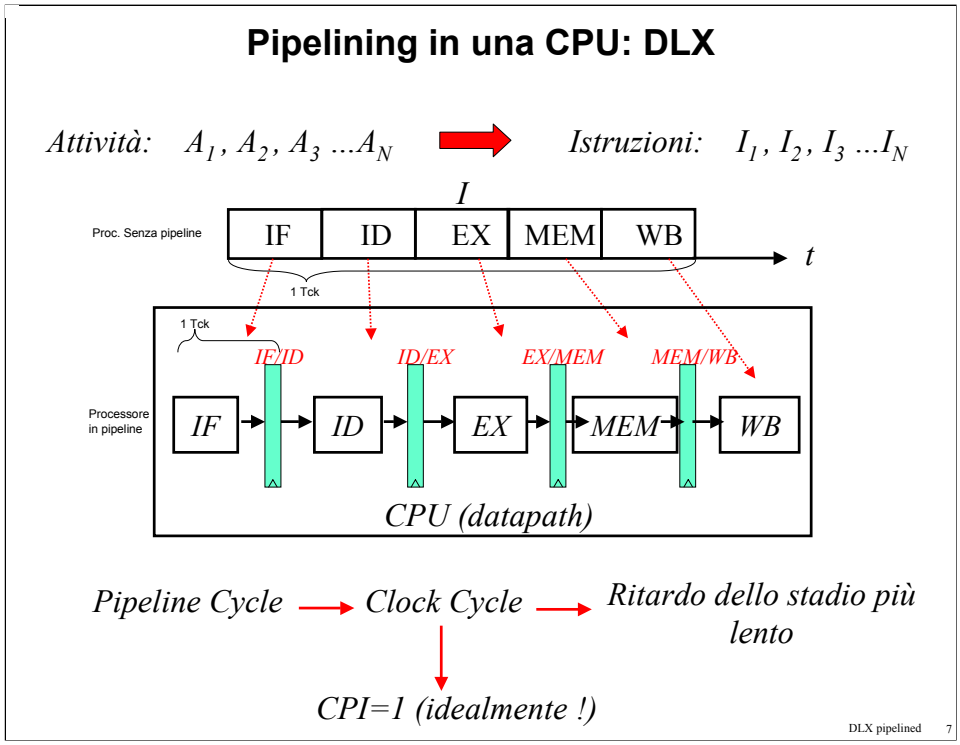
Nel caso ideale, con pipeline perfettamente bilanciata, lo speedup è esattamente = K

Nella pratica, il progettista della pipeline potrà soltanto cercare di tendere a questo risultato: infatti non è possibile ottenere una pipeline rale perfettamente bilanciata perché non è possibile suddividere l'attività da svolgere in passi caratterizzati dallo stesso tempo di esecuzione: in generale i passi avranno tempi di esecuzione diversi e, poiché la pipeline è un sistema perfettamente sincrono in cui è possibile far avanzare le attività in esecuzione solo quando tutti gli stadi hanno terminato, il cycle time Tp sarà determinato dal tempo di esecuzione dello stadio più lento con conseguente perdita sullo speedup calcolato rispetto al caso sequenziale.

Una pipeline reale quindi ha degli stadi caratterizzati da tempi di esecuzione diversi e procede con un "ritmo" che è determinato dallo stadio più lento. Dovremo quindi accettare una situazione di bilanciamento imperfetto, che implica una riduzione dell'efficienza nel sistema: alcuni stadi che avrebbero terminato l'esecuzione del proprio passo devono attendere che lo stadio più lento abbia terminato l'esecuzione, prima di poter riprendere a lavorare. La conseguenza di ciò è che lo speedup ottenibile con una pipeline reale è sempre minore di quello ideale (che è pari al numero degli stadi).

Nell'esempio si assume che l'istruzione A richieda 20t per essere eseguita in una CPU senza pipeline. Con una pipeline a 4 stadi, l'istruzione eseguita (con pipeline a regime cioè con pipeline già piena, ad es. con istr. A eseguita ripetutamente) implica i tempi di esecuzione per ciascuno stadio mostrati nel lucido. Tra questi lo stadio 3 è il più lento con 6t. Visto che il tempo di esecuzione è pari al Numero di attività / throughput, allora N (che è lo stesso per i due fattori) si semplifica e lo spedup è esprimibile come il rapporto dei throughputs. Lo speedup risulta essere il rapporto fra Ta e Tp che sono i denominatori dei due tempi di esecuzione di una istruzione visti nel lucido precedente.

Una pipeline è bilanciata quando le latenze (durata della esecuzione) in ciascuno stadio sono identiche e la loro somma (calcolata senza considerare i ritardi dei pipeline regs) non supera la latenza della esecuzione sequenziale.



Finora abbiamo trattato le basi teoriche per comprendere il concetto di pipeline e identificare le grandezze che la caratterizzano. Adesso applichiamo questo concetto ad al microprocessore DLX.

Introdurre la pipeline in una CPU significa principalmente suddividere in PASSI l'elaborazione di un'istruzione (fetch, decode, execute, memory, writeback) e introdurre dei registri latch tra ciascuno di questi passi, in modo che, attraverso opportuni collegamenti, i risultati parziali di ciascun passo possano essere utilizzati dallo stadio successivo. I 5 passi in cui è stata scomposta l'elaborazione di un'istruzione assembler sono quelli già visti per la descrizione più dettagliata del modello di Von Neumann.

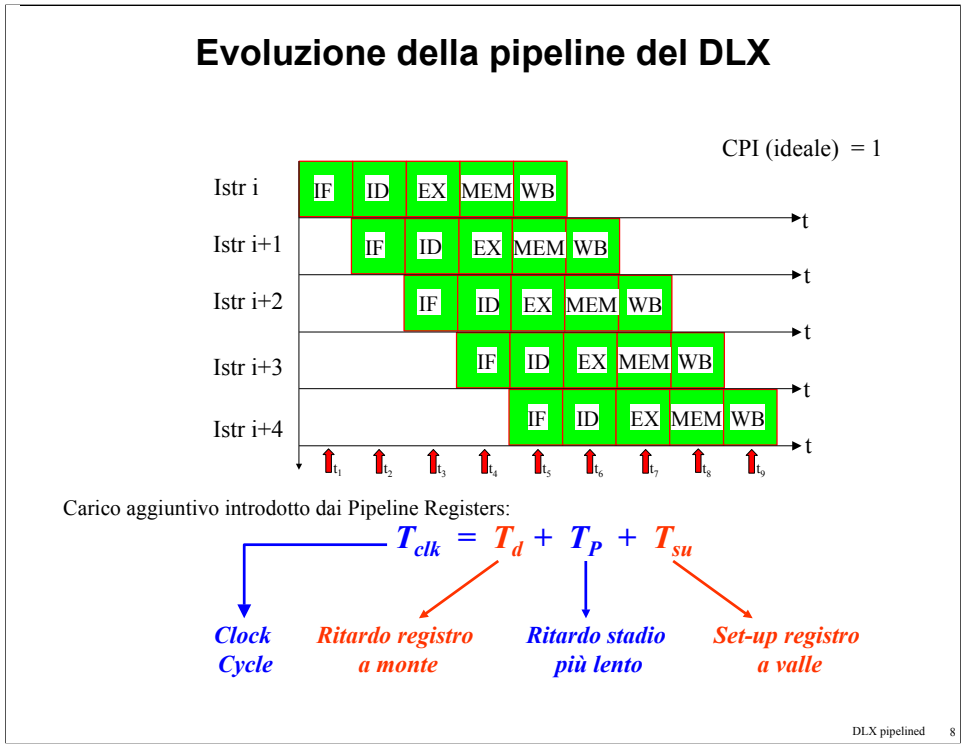
Nello schema del datapath i registri latch appaiono come delle barre divisorie tra i vari stati: questa immagine è stata usata per suggerire che l'elaborazione di un'istruzione è confinata ad uno stadio per ogni ciclo di pipeline: all'avvento del clock, l'istruzione oltrepassa il registro "divisore" e va nello stadio successivo. I registri prendono il nome dai due stadi in mezzo ai quali si trovano.

Nel DLX senza pipeline serve 1Tck per eseguire i compiti dei 5 stadi, mentre su quello in pipeline serve 1Tck per ogni stadio. Sembra quasi che la pipeline non migliori le prestazioni del DLX. Difatti, in generale, la pipeline aumenta il numero di cicli di clock richiesti per eseguire un programma, in quanto alcune istruzioni restano nella pipeline in attesa che siano eseguite le istruzioni che generano i loro input (bolle o stalli di pipeline). Il guadagno delle prestazioni attribuibile alla pipeline deriva dalla minor logica che deve essere eseguita in un ciclo di clock: grazie a questo, un processore in pipeline può ridurre il periodo del ciclo Tck rispetto all'implementazione senza pipeline dello stesso processore. Poiché un processore in pipeline ha un trhoughput di un'itruzione/ciclo, il numero totale di istruzioni eseguite nell'unità di tempo è maggiore nel processore in pipeline e questo migliora le sue prestazioni.

Il Tck di un processore in pipeline dipende da 4 fattori: Tck della versione senza pipeline dello stesso processore, numero di stadi della pipeline, grado di uniformità in cui la logica del datapath è distribuita tra gli stadi della pipeline, la latenza dei latch della pipeline. Se la logica può essere uniformemente ripartita tra gli stadi della pipeline, il periodo di clock del processore in pipeline è dato dalla seguente espressione (ogni stadio contiene la stessa frazione di logica originale + un latch di pipeline):

$$TempoDiCiclo_{InPipeline} = \frac{TempoDiCiclo_{SenzaPipeline}}{NumeroStadi} + LatenzaDelLatch$$

Aumentando il numero degli stadi, la latenza dei latch in pipeline diventa una frazione sempre più grande del tempo di ciclo, limitando il vantaggio di ripartire il processore in un numero molto grande di stadi. E' possibile ottenere una frequenza di clock un po' più elevata sfruttando il fatto che una pipeline a *n* stadi richiede solo *n-1* latch e assegnando una logica aggiuntiva all'ultimo stadio della pipeline per rendere la sua latenza totale pari a quella degli altri stadi che includono i latch.



La prima istruzione comincia ad essere eseguita a partire dal tempo t1 in cui viene prelevata dalla ram (IF). Gli altri stadi sono tutti vuoti.

Poi al tempo t2 l'istruzione i passa in ID, mentre la seconda istruzione i+1 è prelevata dalla memoria. Gli altri stadi EX, MEM e WB sono vuoti: questo stato delle cose si può leggere scorrendo la colonna degli stadi nel diagramma, dall'alto verso il basso, in corrispondenza dell'istante t2.

Allo stesso modo, in t3 si nota che l'istruzione i è in EX, la i+1 è in ID e la i+3 è in IF, mentre MEM e WB sono ancora vuoti.

E così via fino a T5 in cui la pipeline è "a regime" perché tutti i suoi stadi sono occupati. In presenza di altre istruzioni da eseguire dopo i+4 si mantiene lo stato di regime, durante il quale, ad ogni ciclo di pipeline (≅ Tclock), essa conclude l'esecuzione di una istruzione raggiungendo il valore ideale di CPI=1.

Rispetto al caso sequenziale bisogna considerare che l'aggiunta dei registri di pipeline comporta il rispetto dei tempi di propagazione e setup da tenere in considerazione nella stima del clock cycle (Tp). Esso dipenderà quindi ancora dal tempo impiegato dall'elaborazione nello stadio più lento, ma anche dal ritardo del registro precedente e dal tempo di setup del registro successivo (questi ultimi si considerano uguali per tutti i pipeline regs).

Requisiti per l'implementazione in pipeline

- 1) Ogni stadio deve essere attivo in ogni ciclo di clock.
- 2) E' necessario incrementare il PC in IF (invece che in ID).
- 3) E' necessario introdurre un ADDER (PC <-- PC+4) nello stadio IF.
- 4) Sono necessari due MDR (che chiameremo MDR e SMDR) per gestire il caso di una LOAD seguita immediatamente da una STORE (sovrapposizione di WB su registro e MEM su ram).
- 5) In ogni ciclo di clock devono poter essere eseguiti 2 accessi alla memoria (IF, MEM) riferiti a Instruction Memory (IM) e Data Memory (DM) (architettura *"Harvard")*.
- 6) Il clock della CPU è determinato dallo stadio più lento: IM, DM devono essere delle memorie cache (on-chip)!
- 7) I Pipeline Registers trasportano sia dati sia informazioni di controllo (il controller è "distribuito" fra gli stadi della pipeline).

DLX pipelined 9

1) è ovvio perché altrimenti si creano delle situazioni in cui uno stadio ha eseguito la sua parte in un Ck e non può passarlo al successivo che non è attivo

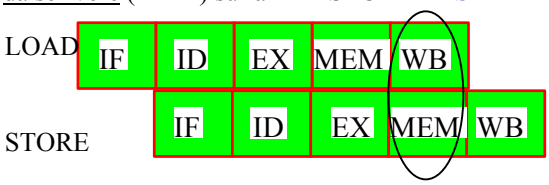
2) l'incremento deve essere fatto nella fase di fetch perché al clock successivo, lo stadio di fetch deve già essere pronto ad eseguire il fetch dell'istruzione successiva senza dover attendere il completamento di ID.

3) Non può più essere la ALU ad occuparsi dell'incremento del PC perché sarà impegnata nella EX di un'altra istruzione: servirà un sommatore ad hoc (ADD) per l'incremento di PC.

4) La risorsa MDR è duplicata rispetto al datapath sequenziale per consentire di possedere, in un dato Ck, entrambe le info, ad esempio nel caso di una LOAD seguita da STORE in cui si sovrappongono WB(LOAD) e MEM(STORE) che accederebbero entrambe all'MDR per due contenuti diversi:

• già letta da ram da destinarsi (WB) al registro ➔ LOAD ➔ MDR = loaded Memory Data Register

• da scrivere (MEM) su ram ➔ STORE ➔ SMDR = to be Stored Memory Data Register



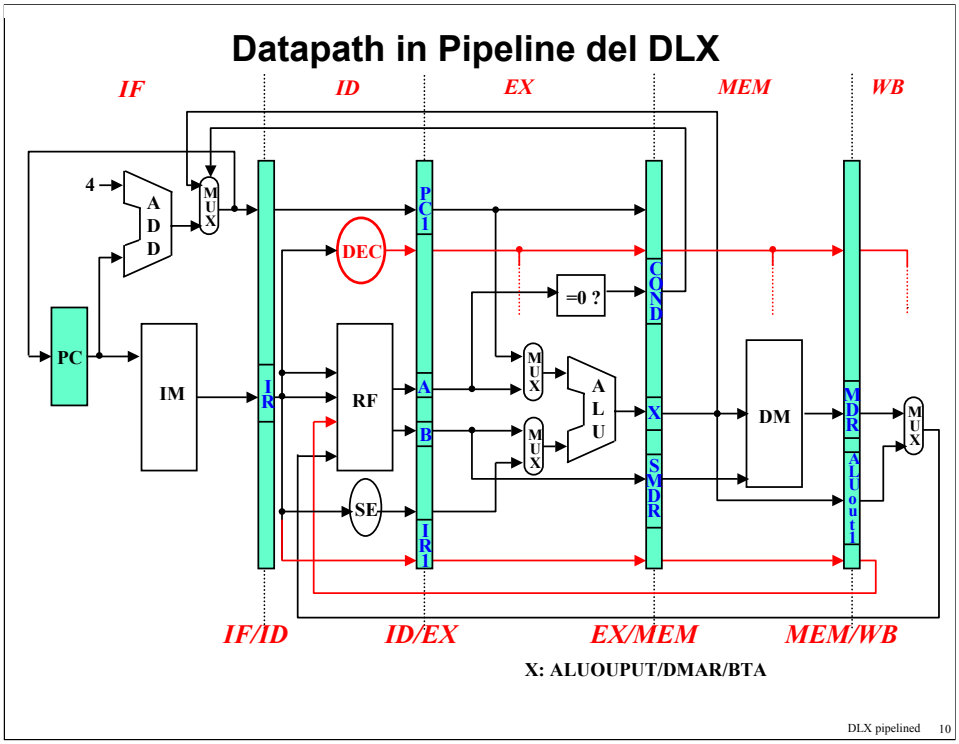
Per questa istr, MDR deve contenere il dato letto dalla mem

Per questa istr, MDR deve contenere il dato che deve essere scritto in memoria

NON PUO' TRATTARSI DELLO STESSO REGISTRO

5) i due accessi alla memoria contemporanei per il codice e i dati in ciascun ciclo di clock sono possibili solo se ci si serve di due memorie *cache* separate per dati e istruzioni. Una memoria cache contiene una copia di una porzione di ram (fino a centinaia di KB) e che, basandosi sulle ipotesi di località spaziale e temporale degli accessi in memoria, consente alla CPU di accedere per molti dati e molte istruzioni in cache (hit) anziché dover leggere il dato sulla ram attraverso il bus di sistema. Le memorie cache sono memorie di per sé veloci e il loro accesso è reso ancor più rapido dal fatto che godono di un indirizzamento privilegiato dato che si trovano sullo stesso PCB della ALU, se non addirittura sullo stesso chip (on-chip). Il loro contenuto viene aggiornato da e verso la porzione di ram replicata, ogni volta che ciò si rende indispensabile. Esistono apposite reti logiche che controllano l'insorgere di tale necessità o del bisogno di copiare in cache una nuova porzione di ram perché l'indirizzo cercato non è in cache (miss).

7) Le linee di controllo sono per lo più omesse negli schemi che vedremo ma non bisogna dimenticare che svolgono un ruolo di primaria importanza. Il fatto che il Controller sia distribuito tra gli stadi dipende dal fatto che le azioni che si svolgono in ogni stadio possono dipendere dallo stato di ciascun altro singolo stadio (anche quelli successivi!) che quindi deve tenere memoria del dato e dei conseguenti controlli a beneficio di tutti gli altri stadi che da esso dovessero dipendere.



I passi in cui viene suddivisa l'esecuzione di un'istruzione DLX sono quelli già noti IF, ID, EX, MEM, WB: ogni passo dell'esecuzione di un'istruzione viene eseguito da un'unità funzionale detta stadio; per consentire agli stadi sia di operare simultaneamente su istruzioni diverse sia di scambiarsi informazioni tra essi, vengono inseriti dei registri, detti pipeline registers. Per semplicità pensiamo che siano dei registri edge-triggered che campionano sui fronti del clock della CPU, anche se è possibile utilizzare dei latch ed una diversa strategia di clocking.

In un generico ciclo di clock ogni stadio elabora le informazioni presenti sul suo registro di ingresso, relative ad un'istruzione (diversa da stadio a stadio) e produce delle nuove informazioni e controlli, anch'esse relative all'istruzione. All'inizio del ciclo successivo queste informazioni e i controlli conseguenti vengono campionati sul registro d'uscita dello stadio, che è il registro d'ingresso dello stadio successivo: in questo modo lo stadio successivo esegue il passo che gli compete dell'istruzione che era nello stadio precedente e tutte le istruzioni avanzano di un posto. In sostanza, grazie ad i registri di pipeline, in un dato ciclo ogni stadio lavora su informazioni relative ad un'istruzione diversa, e le istruzioni presenti nella pipeline avanzano di uno stadio ad ogni clock.

L'uso dei pipeline regsister non è specifico del datapath dl DLX: ogni CPU in pipeline usa questi registri per trasportare le informazioni relative ad una sequenza di istruzioni attraverso gli stadi della pipeline. Vediamo invece ora cosa accade negli stadi della pipeline del DLX.

I primi due stadi sono indipendenti dal tipo di istruzione, mentre il tipo di elaborazione che viene eseguito nei 3 stadi successivi dipende dal tipo di istruzione. Considereremo le istruzioni ALU, MEM (LOAD, STORE) e BRANCH.

IF: Il PC fornisce l'indirizzo usato per accedere alla IM e prelevare l'istruzione: simultaneamente il PC viene incrementato al fine di ottenere l'indirizzo che sarà usato per prelevare la prossima istruzione. Ignoriamo per ora la presenza del MUX.

ID: l'istruzione prelevata nello stadio IF viene campionata su IF/ID e decodificata nello stadio ID: la rete di decodifica genera i segnali di controllo che vengono propagati verso tutti gli stadi attraverso i registri: in ID il decoder genera i segnali che verranno poi utilizzati nei vari stadi man mano che l'istruzione procede nella pipeline. In ID viene anche letto un eventuale displacement di un salto.

Quasi tutte le istruzioni del DLX prevedono l'accesso in lettura ad uno o due registri: i campi dell'istruzione che specificano i registri da leggere occupano una posizione fissa nell'istruzione e nettamente separata dall'opcode: è possibile allora utilizzare questi campi per accedere ai regsitri e prelevare gli operandi dell'istruzione simultaneamente alla fase di decodifica: leggiamo gli operandi mentre siamo decodificando l'istruzione, senza sapere bene ancora cosa dovremo fare: poco male, se ci servirà un solo operando, o nessuno non li useremo, ma non ci costa nulla leggerli. I tre stadi successivi sono dipendenti dall' istruzione:

EX(ALU): dobbiamo eseguire un'operazione logico/aritmetica sui registri letti in ID: i MUX selezioneranno come ingressi della ALU i valori dei registri provenienti dal regsitro ID/EX i segnali di controllo della ALU, provenienti dallo stadio precedente, selezioneranno l'operazione da eseguire: il risultato sarà pronto all'uscita della ALU alla fine del ciclo di clock e campionato sul registri EX/MEM all'inizio del clock successivo.
EX(LOAD/STORE): lo stadio EX è usato per calcolare l'indirizzo per renderlo disponibine in ingresso a EX/MEM: ricordando che la modalità di indirizzamento è displacement, e che il registro che contiene la base dell'indirizzo è già stato letto in ID, quello che è necessario fare è abilitare la via in basso del MUX e come secondo operando abilitare il displacement: questo è contenuto nell'istruzione e viene estratto e convertito a 32 bit nello stadio ID e passato allo stadio EX mediante il registro di pipeline ID/EX. **EX(BRANCH)**: viene fatto il check della condizione ed il valore è disponibile per EX/MEM e la ALU calcola anche il BTA come displacement del PC+4 calcolato precedentemente dal sommatore che ha un collegamento con la ALU attraverso i pipeline regs.

Quindi dopo lo stadio EX il registro EX/MEM contiene il risultato dell'elaborazione, se è un'istruzione ALU, l'indirizzo se è una LOAD (o un BRANCH?).

MEM(ALU): non bisogna accedere alla memoria: il risultato verrà semplicemente passato allo stadio WB attraverso il registro di pipeline MEM/WB. **MEM(LOAD)**, si accede alla memoria tramite l'indirizzo presente su EX/MEM ed il dato letto viene campionato poi nel registro MEM/WB. **MEM(STORE)**, accesso in mem con indirizzo come prima, e da EX/MEM proviene anche il dato scritto. **MEM(BRANCH)**: se COND=true c'è branch completion cioè il MUX porta il BTA all'ingresso del PC perché nel clock dopo si deve fare il fetch dell'istruzione che sta all'ind del salto

WB: è attivo solo per istr di tipo ALUo LOAD. In entrambi i casi bisogna scrivere sul Register File. **WB(ALU)**: il valore da scrivere è quello proveniente dal Data Memory. **WB(LOAD)**: il valore è quello che è stato propagato dallo stadio MEM: il MUX in WB consente di selezionare fra i due valori quello che viene inviato sulla porta di scrittura del RF. L'identificatore del registro che deve essere scritto (campo Rd dell'istruzione) viene anch'esso dallo stadio WB, su cui è stato propagato da ID: difatti non possiamo usare il campo Rd dell'istruzione correntemente in ID, perché così facendo scriveremmo erroneamente sulla destinazione dell'istruzione correntemente in ID che non è quella di cui stiamo seguendo l'evoluzione attraverso la pipeline, bensì quella che la segue di 3 clock (è stata prelevata con 3 clock di ritardo). **WB(BRANCH)**: non si fa nulla perché il branch completion è stato fatto in MEM. **WB(STORE)**: non si fa nulla.

Legenda: IF Instruction Fetch, ID Instruction Decode, EXecution, MEMory, WB WriteBack, PC Program counter, IM Instruction Memory, DECoder di istruzione, ALU Arithmetic Logic Unit, DM Data Memory, ALUOUTPUT registro di pipeline dell'uscita della ALU in EX/MEM,

Esecuzione in pipeline di un'istruzione "ALU"

IF	IR←M[PC] ; PC←PC+4
ID	A←RS1; B←RS2; PC1←PC; IR1←IR (per gli stadi successivi)
EX	ALUOUTPUT←A *op* B oppure ALUOUTPUT←A *op* $IR_{0..15}$ ## $(IR_{15})^{16}$
MEM	ALUout1←ALUOUTPUT ("parcheggio" in attesa di WB)
WB	RD←ALUout1

Esecuzione in pipeline di un'istruzione "MEM" (Load o Store)

IF	IR ←M[PC] ; PC←PC+4
ID	A←RS1; B←RS2; PC1←PC;IR1←IR
EX	DMAR←A *op* $IR_{0..15}$ ## $(IR_{15})^{16}$ SMDR←B
MEM	MDR←M[DMAR] ***(LOAD)*** oppure M[DMAR]←SMDR ***(STORE)***
WB	RD←MDR ***(LOAD)***

Esecuzione in pipeline di un'istruzione "BRANCH"

IF	IR ← M[PC]; PC ← PC+4
ID	A ← RS1; B ← RS2; PC1 ← PC; IR1 ← IR
EX	BTA ← PC1 + $IR_{0..15}$ ## $(IR_{15})^{16}$ Cond ← A *op* 0
MEM	if (Cond) PC ← BTA
WB	- - - - - -

BTA = BRANCH TARGET ADDRESS
indirizzo del salto (calcolato in ID)

Alee nelle Pipeline

Si verifica una situazione di "*Alea*" (h*azard*) quando in un determinato ciclo di clock un'istruzione presente in uno stadio della pipeline non può essere eseguita in quel clock.

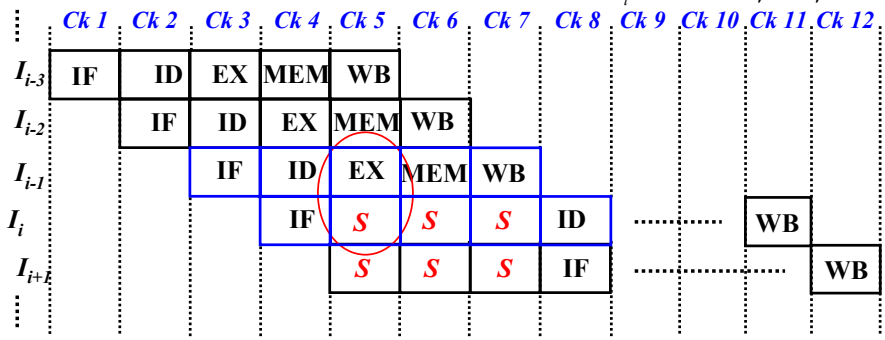
- *Alee Strutturali* - Una risorsa è condivisa fra due stadi della pipeline: le istruzioni correntemente in tali stadi non possono essere eseguite simultaneamente.
- *Alee di Dato* – Sono dovute a *dipendenze* fra le istruzioni. Ad esempio un istruzione tenta di leggere un registro prima che l'istruzione precedente l'abbia scritto (*RAW)*.
- *Alee di Controllo* – Le istruzioni che seguono un branch *dipendono* dal risultato del branch (*taken/not taken*).

L'istruzione che non può essere eseguita viene bloccata ("stallo della pipeline"), insieme a tutte quelle che la seguono, mentre le istruzioni che la precedono avanzano normalmente (così da rimuovere la causa dell'alea). Lintroduzione degli stalli è una possibile soluzione al problema delle alee

DLX pipelined 14

Alee e Stalli

Caso di un'alea di dato RAW nel DLX: I_{i-1} = `ADD R3, R1, R4`
I_i = `SUB R7, R3, R5`



$T_5 = 8 * CK = (5 + 3) * CK$

$T_5 = 5 * (1 + 3/5) * CK$

CPI ideale — Stalli per istruzione

generalizz.

$T_N = N * 1 * CK$

$T_N = N * (1 + S) * CK$

CPI effettivo

DLX pipelined 15

Impatto degli stalli sullo Speedup

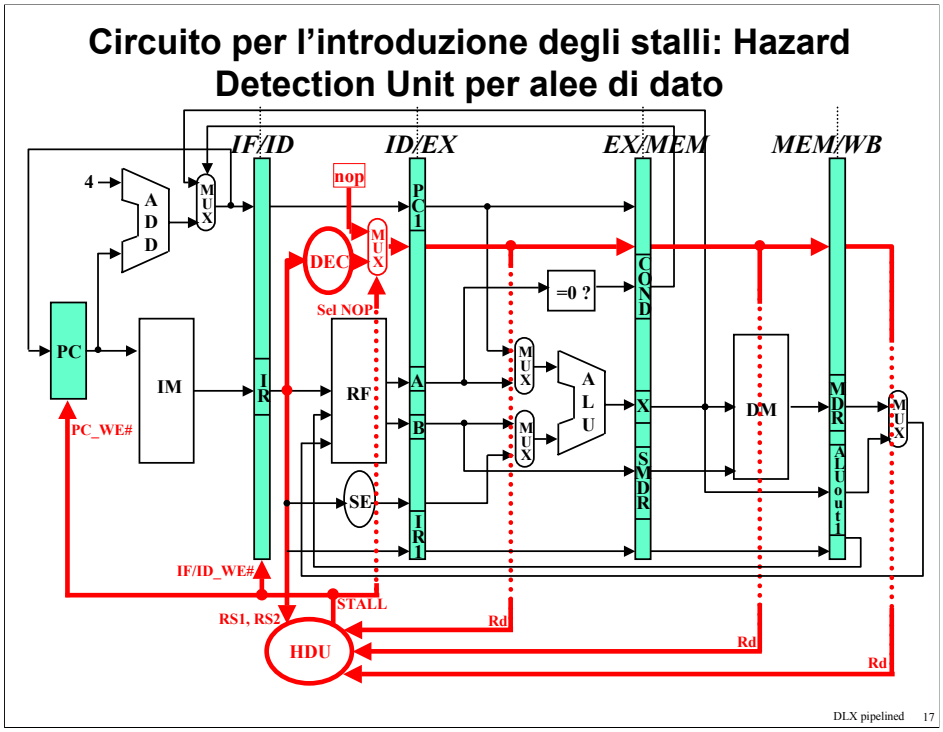
s: non-pipelined
p: pipelined

$$Speedup = \frac{T_N(s)}{T_N(p)} = \frac{N * CPI(s) * Ck(s)}{N * CPI(p) * Ck(p)} = \frac{CPI(s)}{1+S} * \frac{Ck(s)}{Ck(p)}$$

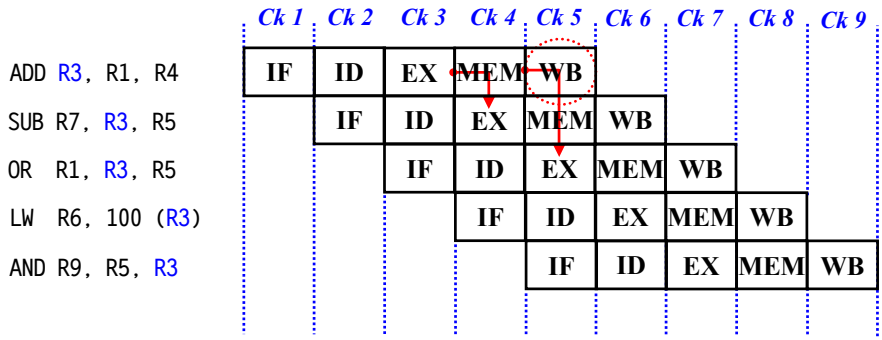
Ipotizzando che i clock siano identici: $Speedup = \frac{CPI(s)}{1+S}$

Ipotizzando che nell'implementazione sequenziale *tutte* le istruzioni impieghino un numero di clock pari al numero degli stadi della pipeline (indicato con K):

$$Speedup = \frac{K}{1+S}$$


Circuito per l'introduzione degli stalli: Hazard Detection Unit per alee di dato
IF/ID
ID/EX
EX/MEM
MEM/WB
4
ADD
MUX
nop
DEC
Sel NOP
PC
IM
IR
RF
A
B
SE
PC1
IR1
=0 ?
COND
X
SMDR
ALU
DM
MDR
ALUout1
PC_WE#
IF/ID_WE#
RS1, RS2
STALL
HDU
Rd
Rd
Rd
DLX pipelined 17

Forwarding

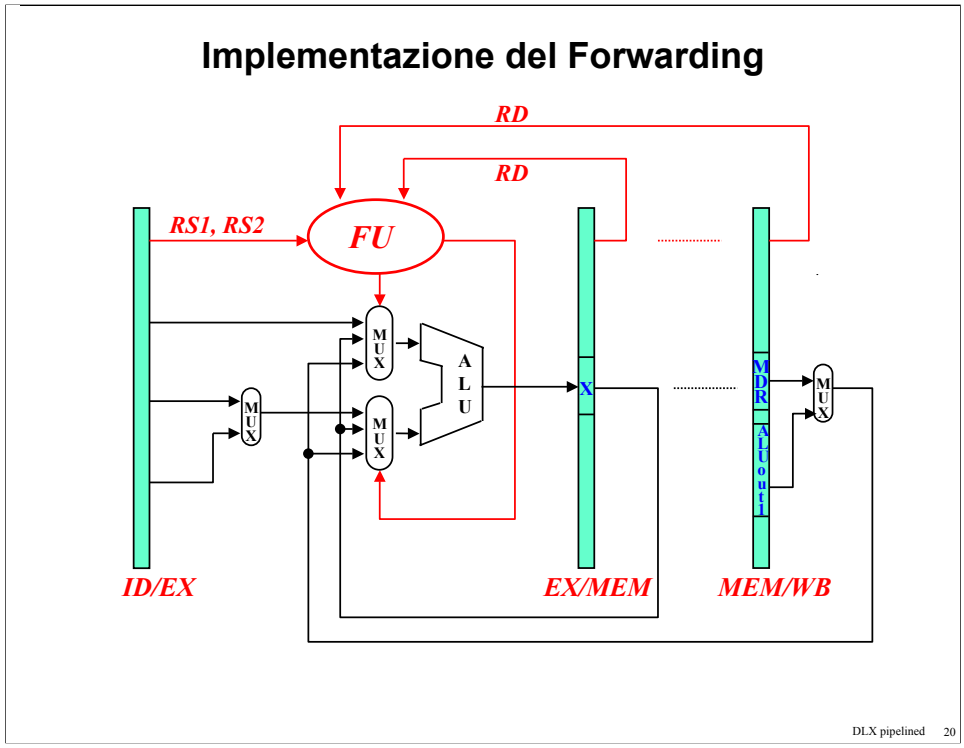


Se il RF ritorna il nuovo valore in caso di RD/WR sullo stesso registro non c'è alea fra la AND e la LW.

Il *forwarding* consente di eliminare quasi tutte le alee di tipo RAW della pipeline del DLX *senza stallare* la pipeline.

Esercizio

• Si disegni lo schema logico del Register File del DLX nell'ipotesi che esso debba restituire il nuovo valore in caso di lettura/scrittura simultanea sullo stesso registro.



Riprendiamo il discorso sul forwarding introducendo ora uno schema di principio per la sua implementazione. I due collegamenti all'indietro si inseriscono sui due MUX a cui arrivano all'ALU anche gli operandi A e B.

E' necessario anche prevedere la Forwarding Unit che controlla questi due MUX che decidono cosa va all'ALU: se i valori da A e/o B oppure i valori anticipati (forwarded) recuperandoli dagli stadi seguenti.

Alea di dato dovuta alle istruzioni di LOAD

LW R1,32(R6)	IF	ID	EX	MEM	WB	
ADD R4,R1,R7		IF	ID	EX	MEM	
SUB R5,R1,R8			IF	ID	EX	
AND R6,R1,R7				IF	ID	

N.B. il dato richiesto dalla ADD è presente solo alla fine di MEM. L'Alea non può essere eliminata con il Forwarding!

E' necessario stallare la pipeline

LW R1,32(R6)	IF	ID	EX	MEM	WB	
ADD R4,R1,R7		IF	ID	*Stall*	EX	MEM
SUB R5,R1,R8			IF	*Stall*	ID	EX
AND R6,R1,R7				*Stall*	IF	ID

DLX pipelined 21

Ora vediamo il caso di una ADD che deve usare un dato caricato da memoria dall'istruzione LOAD precedente. In pipeline la ADD usa il dato di R1 nel passo EX, proprio mentre la LOAD lo sta caricando dalla memoria: non è ancora disponibile. Non si può pensare a qualche tipo di forwarding perchè il dato è proprio fisicamente assente dalla CPU e non c'è modo di creare scorciatoie per averlo.

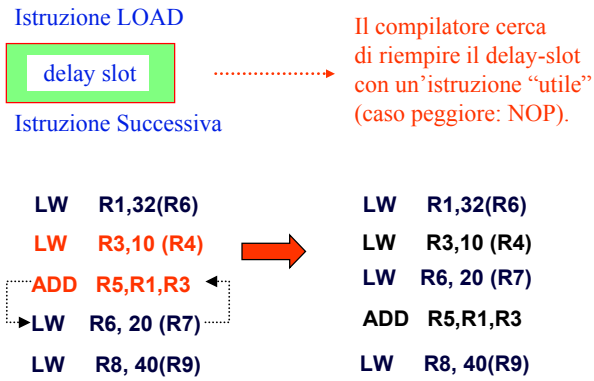
L'unica soluzione torna ad essere l'introduzione di una bolla nella pipeline.

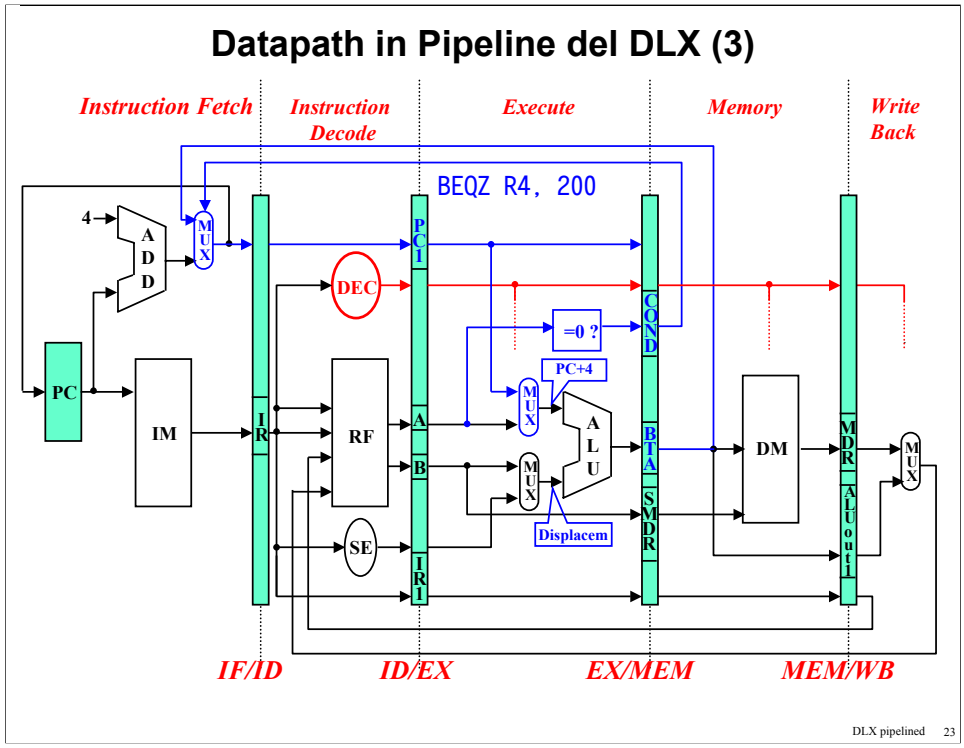
Dopo aver introdotto uno stallo, durante l'EX della ADD il dato non si troverà ancora in R1 ma in MDR (= loaded Memory Data Register), tuttavia adesso con un forwarding da questo registro alla ALU sarà possibile eseguire la ADD pagando il prezzo di un solo stallo.

Si dice anche che nel DLX non è implementato il forwarding in memoria e questo tipo di alee si risolve solo con uno stallo.

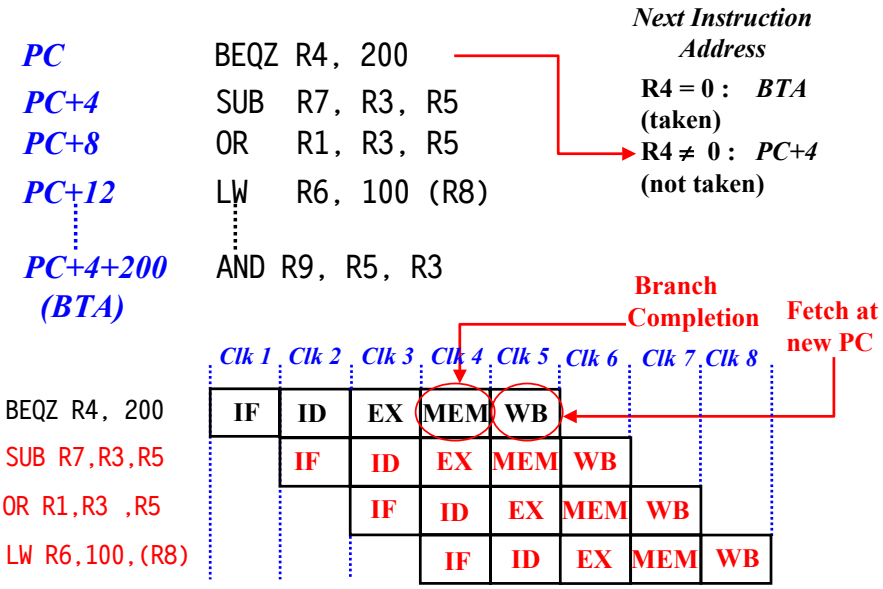
Delayed load

In diverse CPU RISC l'alea associata alla LOAD non è gestita in HW stallando la pipeline ma è gestita via SW dal compilatore (*delayed load*):



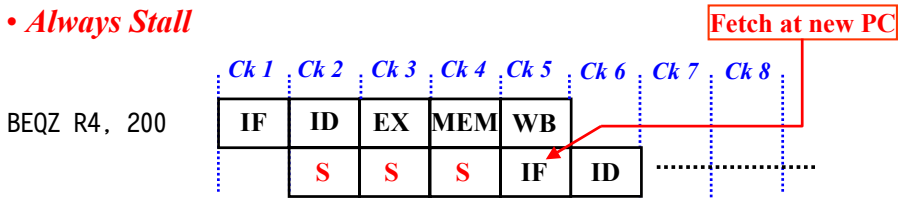

Datapath in Pipeline del DLX (3)
Instruction Fetch
Instruction Decode
Execute
Memory
Write Back
BEQZ R4, 200
4
ADD
MUX
PC
IM
IR
DEC
RF
SE
PCI
A
B
IRI
=0 ?
PC+4
MUX
MUX
ALU
Displacem
COND
BTA
SMDR
DM
MDR
ALUout1
MUX
IF/ID
ID/EX
EX/MEM
MEM/WB
DLX pipelined 23

Alee di Controllo



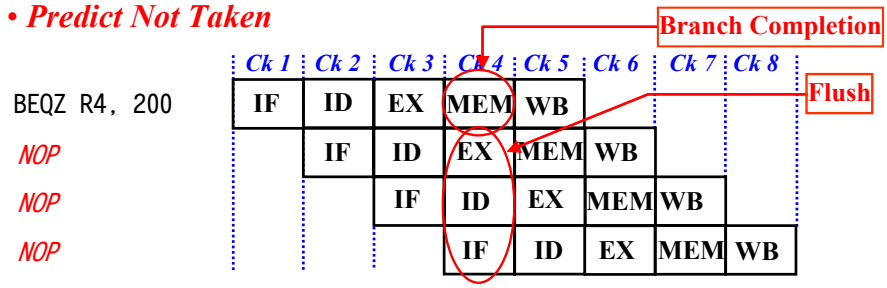
Gestione delle Alee di Controllo (1)

• *Always Stall*



Hyp.: Branch = 25 % ⟶ *CPI = (1 + S) = (1 + 3 * 0.25) = 1.75*

• *Predict Not Taken*

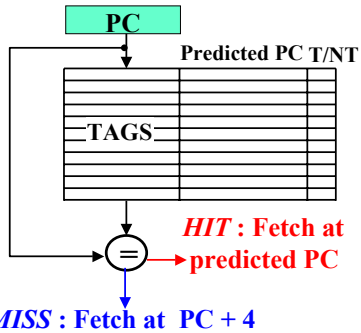


DLX pipelined 25

Gestione delle Alee di Controllo (2)

Hyp.: Branch = 25 %
Taken = 65 % → *CPI = (1 + S) = (1 + 3 * 0.65 * 0.25) = 1.48*

- ***Dynamic Prediction:*** **Branch Target Buffer**



***HIT* : Fetch at predicted PC**

***MISS* : Fetch at PC + 4**

HIT Predizione Corretta : 0 stalli
Predizione Errata : (3 +1) stalli
MISS Predict Not Taken
Taken: (3+1) stalli
Not Taken: 1 stallo

H: Hit Rate, *E*: % di Predizioni Errate
B: % di Branch
T/NT: % di Branch Taken/Not Taken

S = ((H * E * 4 + (1-H) * (T * 4 + NT * 1)) * B

S = ((0.9 * 0.1 * 4 + 0.1 * (0.65 * 4 + 0.35* 1)) * 0.25 = 0.16 CPI=(1 +S) = 1.16

DLX pipelined 26

Delayed branch

• **Similmente al caso della LOAD, in diverse CPU RISC l'alea associata alle istruzioni di BRANCH è gestita via SW dal compilatore (*delayed branch*):**

Istruzione LOAD

delay slot

delay slot

delay slot

Istruzione Successiva

Il compilatore cerca
di riempire i delay-slot
con istruzioni "utili"
(caso peggiore: NOP).

• **Introducendo opportune modifiche al datapath è possibile anticipare l'esecuzione del branch fino a ridurre la penalità ad un solo clock (un solo delay slot, come nel caso della LOAD).**