

# Architetture dei sistemi di elaborazione

## Introduzione ai processori RISC

Parleremo dei processori RISC sono dei processori che hanno un set d'istruzioni molto ristretto. La loro forza è il funzionamento a pipeline che permette di aumentare le prestazioni rispetto ad un processore CISC. Nel MIPS64 abbiamo 31 registri (da 0 a 30) dove il registro 0 è sempre scritto con la costante 0 (non può essere sovrascritto).

Il metodo d'indirizzamento è basato su un accesso immediato in memoria. Bisogna prima sommare un valore nel registro che useremo per l'accesso in memoria ADDUI R1, R2, #32 ( $R1 \leftarrow R2 + 32$ ) oppure ADDUI R1, R0, #32 ( $R1 \leftarrow -32$ ) e poi posso accedere in memoria utilizzando LD R1, 30(R2) ( $R1 \leftarrow \text{MEM}(R2 + 30)$ ). Uso quindi 30 come offset rispetto all'indirizzo che sta in R2 e metto il risultato in R1.

Posso eseguire anche un accesso diretto in memoria. LD R1, 0(R2) sommando così 0 come offset di R2 quindi in realtà accedo direttamente al contenuto puntato dall'indirizzo di R2

Oppure posso fare un accesso del tipo LD R1, 24(R0) per accedere all'indirizzo in posizione 24.

## Formato delle istruzioni

Il formato delle istruzioni nel processore MIPS64 è di lunghezza fissa pari a 32bit.

Le istruzioni che possiamo gestire dalla cmq sono:

- Immediato
- Registro
- Jump

### Instruction Format – Immediate

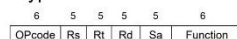
- I – type instruction



Field	Description
<i>opcode</i>	6-bit primary operation code
<i>Rs</i>	5-bit specifier for the source register
<i>Rt</i>	5-bit specifier for the target (source/destination) register
<i>Immediate</i>	16-bit signed <i>immediate</i> used for logical operands, arithmetic signed operands, load/store address byte offsets, and PC-relative branch signed instruction displacement

### Instruction Format – Register

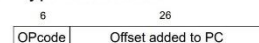
- R – type instruction



Field	Description
<i>opcode</i>	6-bit primary operation code
<i>Rd</i>	5-bit specifier for the destination register
<i>Rs</i>	5-bit specifier for the source register
<i>Rt</i>	5-bit specifier for the target (source/destination) register
<i>Sa</i>	5-bit shift amount
<i>Function</i>	6-bit function field used to specify functions within the primary opcode SPECIAL

### Instruction Format – Jump

- J – type instruction



Field	Description
<i>opcode</i>	6-bit primary operation code
<i>Offset</i>	26-bit index shifted left two bits to supply the low-order 28 bits of the jump target address

Come possiamo vedere, il formato è molto simile. Infatti, tutte le istruzioni contengono i primi 6 bit che sono il codice operativo il quale permette di comprendere l'istruzione in arrivo.

## Instruction Set

Le instruction set del MIPS64 sono:

- Load and Store
- ALU operations
- Branches and Jumps
- Floating Point
- Miscellaneous

Ogni istruzione è su 32 bit. Come possiamo notare sono molte meno dell'istruzione set che ad esempio avevo con 8086.

## Load and Store

L'accesso alla memoria principale è consentito solamente tramite le operazioni di load and store.

Queste sono del tipo:

- Load double word: LD R1, 28(R0) R1<-MEM[28+0]
- Load byte: LB R1, 28(R8)
- ETC

Ogni istruzione in MIPS64 è eseguita in pipeline.

**Il throughput è il numero delle istruzioni che escono dalla pipeline nell'unità di tempo.** Tutti gli stadi della pipeline sono sincronizzati e quindi la velocità del throughput è influenzato dall'operazione più lenta.

In MIPS64 l'esecuzione di ogni istruzione prevede 5 cicli di clock.

- Instruction Fetch Cycle: Il processore prende il PC, lo usa per accedere in memoria e carica la prossima istruzione da eseguire
- Instruction decode/register fetch cycle: Il processore capisce quale istruzione ha davanti e carica gli operandi opportuni nei registri di input della ALU
- Execution/effective address cycle: La ALU calcola l'istruzione partendo dai registri di input, producendo così un output
- Memory access/branch completion cycle (MEM): Scrivo in memoria il risultato calcolato dalla ALU oppure in caso di salto utilizzerò l'indirizzo calcolato dalla ALU e lo scriverò nel PC.
- Write-back cycle (WB): Scrivo il risultato nel registro destinazione

Nella situazione ideale, ovvero quando ad ogni periodo di clock viene eseguita un'istruzione, la pipeline aumenta il throughput del processore in base al numero di stadi della pipeline.

$$\text{Throughput}_{\text{pipelined}} = \text{throughput}_{\text{unpipelined}} * n_{\text{stadi}}$$

Un parametro importante è sicuramente il [CPI](#) (Cicli per Istruzione). Tale parametro indica il numero di cicli di clock necessari al microprocessore per eseguire un'istruzione. È l'inverso del parametro [istruzioni per ciclo](#).

Quindi per l'esecuzione di una generica ADD fra registri A e B devo:

1. Accedo in memoria e faccio i fetch dell'istruzione e la carico nell'Instruction Register
2. Decodifico l'istruzione dell'IR e riconosco che è una ADD fra due i registri, dentro il formato dell'istruzione ci sono gli identificatori dei due registri (A e B).
3. Giro i multiplexer per preparare i registri corretti all'ingresso della ALU. Eseguì la somma dei registri corretti grazie alla ALU. Risultato messo in ALU out.  
Se avessi avuto un salto condizionato, questo modulo avrebbe svolto un **test sulla condizione specificata (zero?)** e di conseguenza avrebbe deciso se saltare o meno in base al verificarsi dell'azione aggiornando PC.
4. Se è un'istruzione di accesso alla memoria si accede alla memoria tramite l'indirizzo contenuto in quel registro oppure se è un'operazione di salto attivo il multiplexer e salvo nel pc l'indirizzo a cui saltare alla prossima istruzione.
5. Scrivo il risultato nel registro di destinazione

Con i vari colori sono evidenziati i vari stadi per processore Mips64

## Architettura a Pipeline

**Tutti gli stadi della pipeline sono sincronizzati e il tempo per eseguire uno step è chiamato ciclo macchina.** Spesso un ciclo macchina corrisponde ad un ciclo di clock.

Per un'architettura a pipeline sono fondamentali dei registri che separano le varie "fette" di gestione dell'architettura a pipeline. Tali registri dividono l'architettura in 5 stati. Questi registri ricevono tutti lo stesso segnale di clock e lavorano in maniera indipendente. Per ogni colpo di clock caricano il loro contenuto nei registri a valle e memorizzano il contenuto dei registri a monte.

Nella versione senza pipeline, i miei registri potrebbero avere clock diverso ma nella pipeline invece i registri sono sincronizzati per lo stesso tempo di clock. Il segnale di clock però nei processori RISC è più basso rispetto ai processori CISC perché è stabilito dallo stadio più lento a causa di alcuni vincoli.

La velocità del modulo operativo più lento condiziona quindi la velocità massima del clock raggiungibile.

## Problemi della pipeline

-Hazards: Esistono 3 tipi di Hazard che riducono di throughput. Sono delle situazioni che impediscono l'esecuzione di un certo stadio durante il colpo di clock designato. Se faccio funzionare la pipeline come l'abbiamo descritta idealmente abbiamo dei risultati non soddisfacenti:

1. Hazard Strutturale: è legato a come è fatta la pipeline. Dentro la pipeline ci sono delle risorse condivise che possono essere "il collo di bottiglia". Stadi diversi possono dover avere accesso a quella memoria. In caso di hazard strutturale un certo stadio si deve fermare e deve andare in stallo. Per sistemare questo problema posso lavorare sul miglioramento delle risorse. Ad esempio, posso avere un Register File che gestisce contemporaneamente 2 accessi in memoria invece che 1. Tale soluzione non è però sempre attuabile. Lo stallò avviene anche quando ho due operazioni che accedono entrambe in memoria, come ad esempio la fase di MEM e la fase di IF.
2. Data Hazard: Ci sono istruzioni che richiedono il dato prodotto dall'istruzione precedente. Tali istruzioni, in un'architettura a pipeline, possono essere sovrapposte e può accadere che un'istruzione chieda il dato prima ancora che questo venga prodotto. In questo caso si generano dei risultati errati rispetto a quelli attesi (ad esempio-> **ADD R1, R2, R3** e poi subito dopo **SUB R4, R1, R5**). In questo caso posso risolvere il problema in due modi:
  - Data Forwarding ->implementando una tecnica di forwarding (o di bypassing): invece di attendere la fase di MEM prendo direttamente il dato richiesto dalla pipeline per inoltrarlo "al volo" al registro prima della ALU adibito al calcolo dell'operazione successiva. La tecnica di data forwarding è complicata perché deve gestire anche uno store temporaneo del dato in alcune occasioni. Siccome non tutti i problemi di hazard di dato possono essere risolti con questa soluzione, certe volte usiamo la prossima soluzione.
  - Stallo: Per certe operazioni dovrei "tornare indietro nel tempo" quindi non utilizzare la tecnica del forwarding. In questi casi mi tocca mandare in stallo una determinata operazione.
3. Control Hazard: Si generano a causa dei salti. In caso di un branch devo infatti svuotare la pipeline perché le istruzioni caricate precedentemente non sono quelle che correttamente dovevo caricare per l'esecuzione del flusso dati. In questo caso perdo molti cicli macchina perché devo iniziare a ricaricare le istruzioni corrette nella pipeline scartando quelle successive. Per evitare questa situazione si usano metodi di predizione dei salti (per i salti condizionati) oppure metodi di previsione dell'indirizzo a cui saltare.

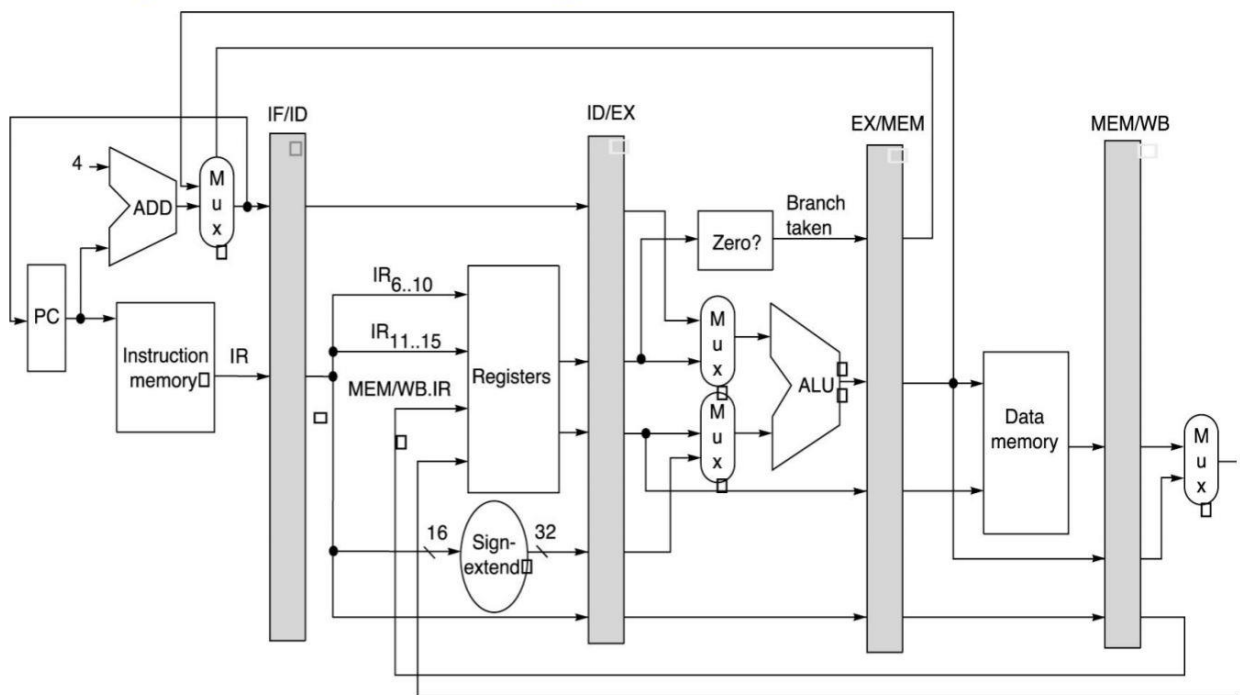
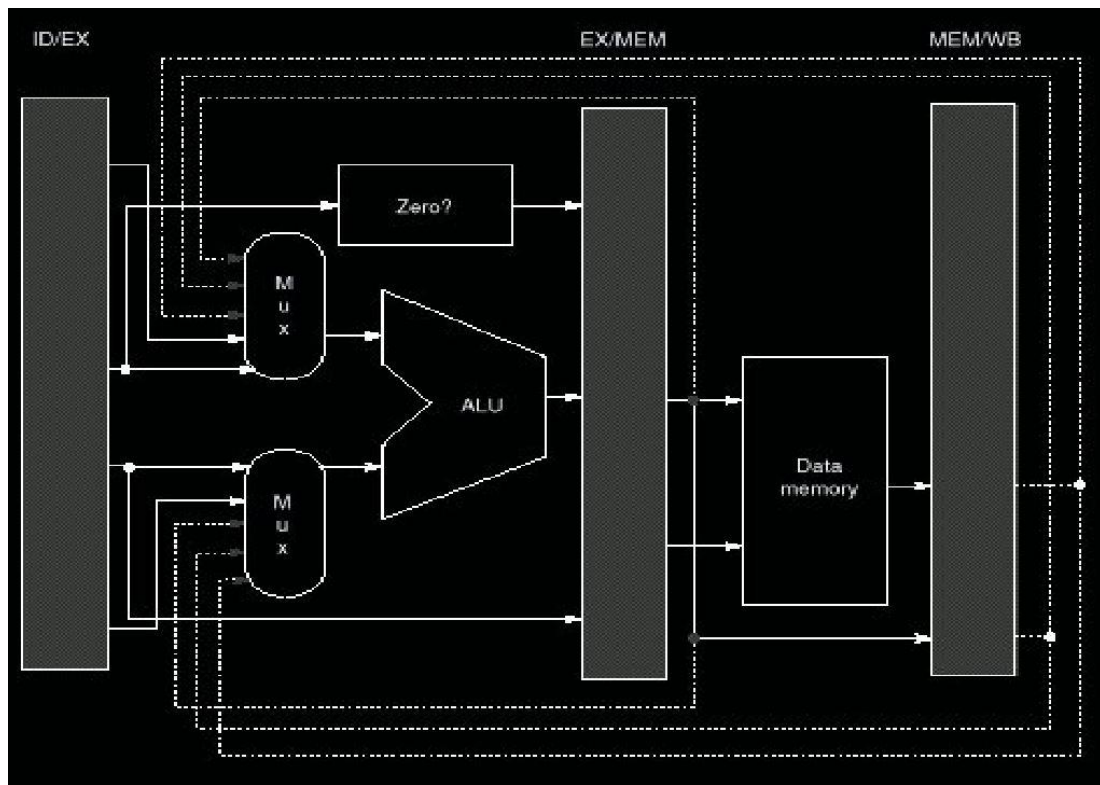
L'unità di controllo deve capire quando usare il data forwarding e quando invece deve utilizzare lo stallo. Ad ogni colpo di clock viene cercato un possibile hazard guardando il contenuto dell'istruzione register che contiene l'istruzione successiva con i relativi registri che saranno adoperati. Tali possibili hazard sono tabulati e l'unità di controllo li conosce. Se l'unità di controllo non trova dipendenze non applicherà nessuna contromisura, in caso contrario dovrà adottare una delle tecniche viste sopra.

Load Interlock Detection			
Situation	Example code sequence		Action
No dependence	LD R1, 45(R2)		No hazard possible because no dependence exists on R1 in the immediately following three instructions.
	DADD R5, R6, R7		
	DSUB R8, R6, R7		
	OR R9, R6, R7		
Dependence requiring stall	LD R1, 45(R2)		Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX.
	DADD R5, R1, R7		
	DSUB R8, R6, R7		
	OR R9, R6, R7		
Dependence overcome by forwarding	LD R1, 45(R2)		Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX.
	DADD R5, R6, R7		
	DSUB R8, R1, R7		
	OR R9, R6, R7		
Dependence with accesses in order	LD R1, 45(R2)		No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.
	DADD R5, R6, R7		
	DSUB R8, R6, R7		
	OR R9, R1, R7		

Il data Hazard può avvenire anche in accesso alla memoria. Se ad esempio eseguo una store e una load che agiscono sulla stessa di memoria e per qualche ragione la store viene rallentata c'è rischio che la load acceda al vecchio valore. È molto più difficile capire a priori un potenziale Hazard in questo caso perché il processore dovrebbe fare un confronto fra gli indirizzi di accesso in memoria della load e della store. In questo caso si sceglie dunque un approccio più conservativo che consiste nell'inserire uno stallo.

Fra i registri a monte della ALU e quest'ultima ci sono dei multiplexer che permettono in hardware di implementare il meccanismo di forwarding. Il data-forwarding consiste quindi nell'aumentare gli ingressi dei multiplexer all'ingresso della ALU.

In caso di detect di un possibile Hazard, l'unità di controllo attiverà delle precise linee del multiplexer in modo da immettere il dato corretto nella ALU tramite il forwarding.



Spesso capita che moduli operativi differenti debbano utilizzare le stesse strutture in modo contemporaneo. In questo caso posso incorrere in degli hazard. Per evitare ciò posso:

- Aumentare le risorse Hardware in modo che permettano più accessi contemporaneamente. Ad esempio, Register-file (write ports) che permettano la scrittura di più registri in modo contemporaneo. Ad esempio, PC nella fase di IF e registri nella fase di WB.
- Introdurre degli stalli per evitare l'accesso concorrente alle stesse strutture fisiche da più stadi della pipeline.

In fase di progettazione bisogna trovare il giusto trade-off fra costo e prestazioni. Implementare un Hardware più elaborato ha un costo giustamente più elevato.

## Analisi degli effetti degli Hazard

Gli Hazard ci creano una diminuzione delle performance all'interno della nostra architettura a pipeline. Ad esempio, un'istruzione di salto mi genera il caricamento dell'istruzione successiva errata nella fase di Instruction Fetch.

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor+1				IF	ID	EX	MEM
Branch successor+2					IF	ID	EX

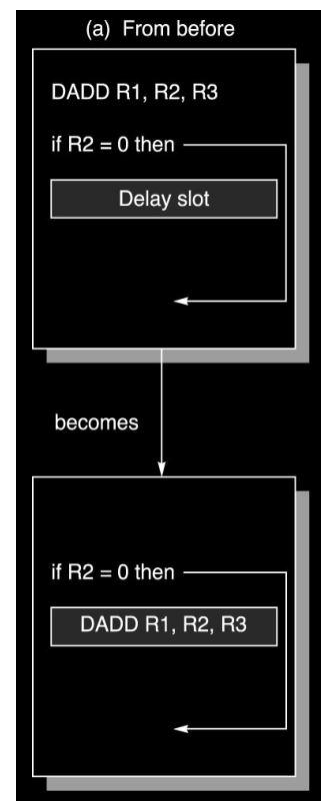
Solo nel Decode mi accorgerò che questa IF è di un'istruzione che non devo prendere.

Sto quindi diminuendo il throughput della mia pipeline.

Per risolvere questo problemi posso agire in diversi modi:

- Freezing della Pipeline: svuoto la pipeline quando incontro un'istruzione di salto condizionato. Caricando le istruzioni corrette. Questa soluzione è la più semplice da implementare.
- Predizioni untaken per i salti condizionati: prevedo i salti condizionati come non presi. Di conseguenza caricherò le istruzioni successive come se il salto non dovesse essere preso e in caso contrario svuoterò la pipeline. Tale soluzione è implementabile se l'hardware lo permette. Il compilatore può generare del codice che massimizza le performance per i branch untaken.
- Predizioni taken per i salti condizionati: prevedo i salti condizionati come presi. Di conseguenza caricherò le istruzioni successive come se il salto dovesse essere preso. In caso contrario svuoterò la pipeline per caricare le istruzioni corrette. Tale soluzione è implementabile se l'hardware lo permette. Il compilatore può generare del codice che massimizza le performance per i branch taken.
- Delayed Branch: Tale tecnica viene eseguita dal compilatore. Consiste nel riordino delle istruzioni posizionandola dopo un'istruzione di salto condizionato. In questo caso l'istruzione verrà sempre eseguita indipendentemente dall'esito dell'istruzione di salto. Il processore non dovrà dunque fare nulla di speciale. Lo spazio dove inseriamo l'istruzione che spostiamo prende il nome di **branch-delay slot**. L'efficienza di questa tecnica dipende dall'abilità del compilatore di trovare le istruzioni migliori da spostare nel branch-delay slot.

Attualmente le architetture a pipeline sono diventate più ricche di stadi comportando così dei branch-delay slot più grandi. I vantaggi del delay branches sono diminuiti infatti oggi molti processori RISC non supportano la tecnica Delayed Branch.





## Eccezioni

Le eccezioni sono eventi che modificano il normale ordine d'esecuzione del programma.

Ci sono due tipi di eccezioni:

- Interrupt (scatenati dai periferici per richiamare il driver di gestione)
- Eventi interni (accesso in memorie violate, oppure errore aritmetico come divisione per 0)

A differenza dei processori CISC, nei RISC la gestione di questi eventi è difficile a causa della sovrapposizione di istruzioni. Questo perché, a causa dell'architettura a pipeline bisognerebbe gestire più istruzioni contemporaneamente. Più interrupt ed eccezioni potrebbero essere scatenati nello stesso momento.

Gli eventi che scatenano le eccezioni interne sono molteplici, ad esempio:

- I/O device request
- accessi in memoria che non sono in memoria principale (Memory management unit scatena page fault perché deve fare una traduzione da indirizzi logici ad indirizzi fisici)
- L'unità di decodifica se trova un'eccezione che non corrisponde ad un'istruzione allora scatena un'eccezione
- Etc..

### ● Possible causes of exceptions are:

- I/O device request
- Operating system call by a user program
- Tracing instruction execution
- Breakpoint (programmer-requested interrupt)
- Integer arithmetic overflow or underflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory accesses
- Memory-protection violation
- Undefined instruction
- Hardware malfunction
- Power failure.

Anche l'accesso alla memoria, avendo un meccanismo di protezione, scatena un'eccezione in caso di accesso errato alla memoria o se c'è un errore di un bit nella lettura di un dato da un bus.

Vi sono anche casi in cui quando la corrente sta andando via, salvo subito il salvabile scatenando un'eccezione.

Le eccezioni possono essere scatenate dall'unità aritmetica, dall'utente, posso anche mascherare certe eccezioni, vi sono alcune eccezioni che richiedono la ripresa del programma interrotto (come gli interrupt esterni o le chiamate del sistema operativo da parte dall'utente [INT 21H]).

Pipeline stage	Cause of exception
IF	Page fault on instruction fetch
	Misaligned memory access
	Memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch
	Misaligned memory access
	Memory-protection violation
WB	None

Concettualmente vogliamo associare l'eccezione ad un'istruzione, eseguire tutte le istruzioni fino a quella di eccezione e poi riprendere il tutto ripristinando le istruzioni precedenti.

Lo stadio di Write Back è l'unico che non può scatenare delle eccezioni.

In caso di eccezioni simultanee da parte stati differenti bisogna servire prima l'eccezione dell'istruzione che logicamente viene eseguita prima.

Un'eccezione può essere:

- Sincrona: quando avviene sempre nella stessa posizione del codice
- Asincrona: avviene poiché generata da un evento esterno
- User requested: sono simili a procedure
- Coerced Exceptions: fuori dal controllo dell'utente
- User Maskable or NotMaskable: l'utente può forzare l'hardware a non rispondere alle eccezioni
- Within vs. between instructions: Le eccezioni possono iniziare durante l'esecuzione delle istruzioni o fra l'esecuzione di due istruzioni
- Resume vs. terminate: Dopo un'eccezione il programma in esecuzione può terminare oppure potrà riprendere dallo stato precedente.

## Gestione delle eccezioni

Vogliamo che l'esecuzione dell'eccezione non interrompa la normale esecuzione del programma. Tale funzionalità è implementata dalle Restartable Machines. Sono delle macchine che permettono di gestire le eccezioni salvando lo stato del programma e riprendendo dal momento precedente all'eccezione. Tutti i processori odierni sono Restartable Machines.

Per interrompere il normale flusso d'esecuzione del programma, la pipeline deve eseguire i seguenti step:

- Forza un'istruzione "trap" nell'IF stage
- Finché la "trap" non è presa blocco la scrittura dell'istruzione che ha generato l'eccezione e di tutte le istruzioni successive
- Quando la procedura che gestisce l'eccezione riceve il controllo, questa salva il valore del PC dell'istruzione che ha generato l'eccezione.

Per riprendere l'esecuzione del programma dopo la procedura di gestione dell'eccezione devo:

- Ricaricare il vecchio valore del PC che mi permetterà di riprendere il normale flusso d'esecuzione del programma.

Le eccezioni che verificano queste condizioni, implementando così una Restartable Machines sono chiamate Precise Exceptions.

## Eccezioni Concorrenti

Durante il funzionamento di un'architettura a pipeline possiamo incorrere in eccezioni generate contemporaneamente da più moduli operativi. Ad esempio, lo stadio di MEM potrebbe scatenare un'eccezione di accesso violato alla memoria e contemporaneamente lo stadio di EX potrebbe scatenare un'eccezione legata all'esecuzione di una data operazione aritmetica.



In questo caso ho un'eccezione di data page fault dello stadio di MEM e un Arithmetic exception generata contemporaneamente dalla fase di EX.

In questo caso dovremmo gestire prima l'eccezione lanciata dalla MEM e poi quella della ADD poiché dobbiamo rispettare l'ordine d'esecuzione del codice.

Per gestire l'ordine delle eccezioni devo:

- Associa un flag d'eccezione ad ogni istruzione nella pipeline
- Setto il flag d'eccezione se un'istruzione causa un'eccezione
- Se il flag d'eccezione è settato l'istruzione non può effettuare operazioni di scrittura. In questo modo evitiamo che l'istruzione generi problemi.
- Quando l'istruzione, con il flag d'eccezione settato, raggiunge lo stadio finale (WB), viene lanciata l'eccezione inserendo un'opportuna istruzione di "trap" nell'IF della pipeline.

Quando un'operazione è completata senza che scateni eccezioni è definita come committed.

Alcune istruzioni modificano lo stato dei registri prima che abbiano effettuato il commit. Ad esempio, se eseguo un'operazione di store con incremento anticipato del registro ma questa andrà in *abort* non riuscirò a fare *undo* dell'istruzione ripristinando il valore precedente del registro.

In questo caso lo stato della macchina sarà alterato poiché non riuscirò a fare roll-back dell'istruzione di aggiornamento dell'indice.

Le istruzioni con aggiornamenti impliciti creano vari problemi:

- Gli aggiornamenti impliciti devono essere salvati e, in caso di eccezione, dobbiamo ripristinare i vecchi valori
- Possono causare Data Hazard
- Rendono più difficile la procedura di riempimento del Delay Slot da parte del compilatore.

## Floating point operations

Non posso generare delle operazioni floating point che vengano gestite in 1 solo colpo di clock. L'unità aritmetica floating point deve per forza utilizzare più colpi di clock.

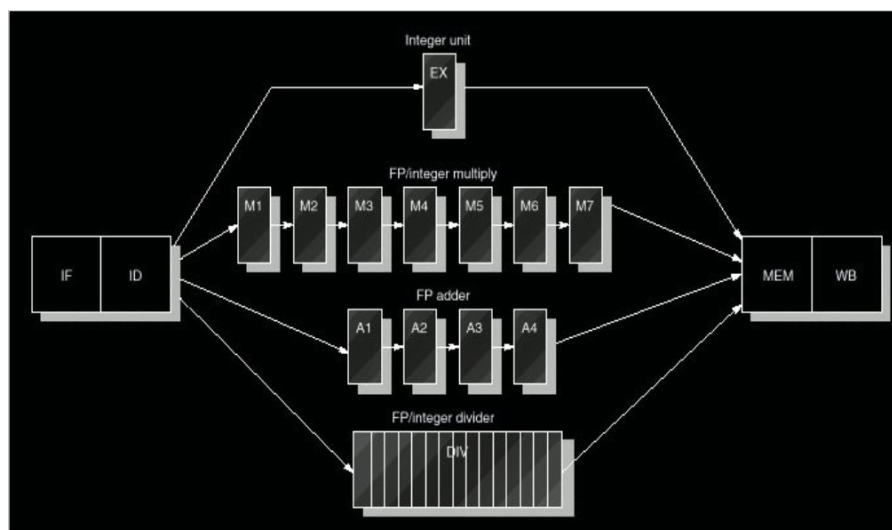
Vi sono delle unità aritmetiche in parallelo che mi permettono di creare una pipeline anche per l'unità aritmetica. Posso quindi iniziare più moltiplicazioni ad ogni colpo di clock anche se per produrre la moltiplicazione ci vogliono più colpi di clock per essere eseguiti. Potrò iniziare una nuova operazione di moltiplicazione floating point ad ogni colpo di clock

I parametri sono:

- **Latenza:** colpi di clock necessari per avere il risultato – tempo che intercorre fra un'istruzione che produce un risultato e un'istruzione che legge il risultato
- **Initiation interval:** È il numero di cicli che devo attendere per processare due istruzioni dello stesso tipo nella stessa unità. Indica quindi il numero di colpi di clock che devono intercorrere fra l'inserimento di un'istruzione e un'altra nella stessa unità. Mi dice se l'unità aritmetica ha un'architettura a pipeline o no. Se sì, posso mandargli dentro una nuova operazione ad ogni colpo di clock, altrimenti devo aspettare un tempo pari ad una latenza per inserire una nuova istruzione. Indica dopo quanti colpi di clock posso inserire un'altra istruzione. Se unità funzionale aritmetica è implementata a pipeline avrò initiation interval = 1.

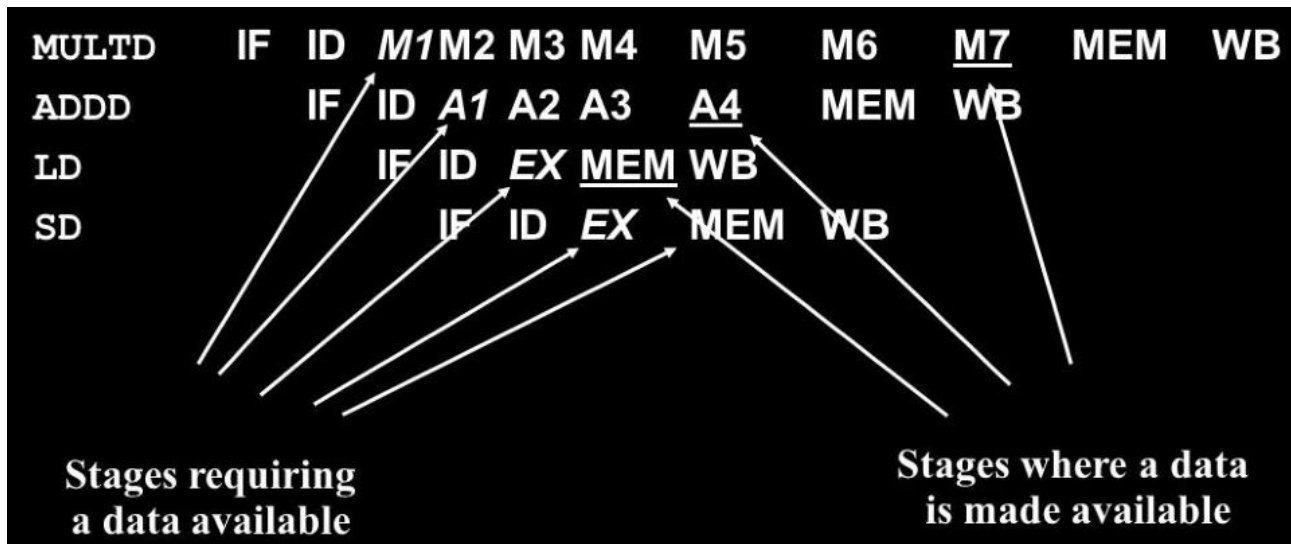
<i>Functional Unit</i>	<i>Latency</i>	<i>Initiation Interval</i>
Integer ALU	0	1
Data Memory	1	1
FP add	3	1
FP/integer multiply	6	1
FP/integer divide	24	25

Alcune unità funzionali aritmetiche possono avere un'architettura a pipeline, altre invece no come la DIV.



Per colpa di queste unità aritmetiche in parallelo (che hanno dimensione di pipeline differente) rischio che l'ordine delle istruzioni in uscita sia differente rispetto a quelle d'ingresso.

Ad esempio, un modulo aritmetico di MUL potrebbe impiegare molti più colpi di clock di una ADD. Posso avere molti hazard strutturali perché vorrei poter accedere al register file con più accessi in simultaneamente. Si devono quindi introdurre degli stalli per gestire questi problemi e le dipendenze di dato. (possiamo risolvere migliorando l'hw, esistono infatti dei register file con più write ports)



Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
MUL.D F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADD.D F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
L.D F2,0(R2)							IF	ID	EX	MEM	WB

Abbiamo il verificarsi di frequenti hazard di dato a causa della latenza. Si verificano quindi degli Hazard di tipo RAW (Read After Write Hazard) che avvengono quando un'istruzione deve attendere in lettura un operando che è in produzione come risultato dall'istruzione precedente. (appunto un'operazione di read dopo un'operazione di write)

Instruction	Clock cycle number																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
L.D F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D F0,F4,F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D F2,F0,F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	WB
S.D F2,0(R2)					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

Qui si generano delle nuove dipendenze di dato (WAW Write After Write Hazard) che è causata dal fatto che le scritture di due dati differenti avvengano in uno stesso registro poiché le istruzioni hanno delle lunghezze differenti. Tale problema si risolve mandando in stallo delle istruzioni. Quindi se un'istruzione sta scrivendo in un registro, prima di un'istruzione precedente, devo mandare in stallo l'istruzione.

Le unità floating point introducono quindi più stalli durante tali operazioni.

**WAW si verifica quando vi sono 2 istruzioni che contemporaneamente nella pipeline scrivono nello stesso registro. Se la seconda delle due scrive prima di un'istruzione precedente ho un WAW.** Causa -> latenza differente fra le istruzioni

Il controllo degli Hazard è effettuato nella fase di Decode (ID stage):

- Hazard Strutturali: Risolti con l'aumento delle write ports
- RAW Hazard: Inserisco degli stalli opportuni in attesa della corretta produzione dell'operando
- WAW Hazard: Effettuo un controllo nella ID stage osservando se vi sono istruzioni con lo stesso operando destinazione. In caso affermativo introdurrò degli stalli.

### **Gestione delle eccezioni in un'architettura floating point – eccezioni imprecise**

Nel caso in cui un'istruzione generi delle eccezioni prima del completamento di un'istruzione a monte devo gestire l'ordine delle eccezioni bufferizzando i risultati. In alcuni casi posso anticipare lo scatenamento delle eccezioni, guardando ad esempio gli operandi (nel caso div per 0).

Ad esempio:

DIV F0, F2, F4  
ADD F10, F10, F8  
SUB F12, F12, F14

L'istruzione di DIV ci metterà molto più tempo per produrre il risultato, durante la sua esecuzione l'istruzione di SUB potrebbe scatenare un'eccezione. Tale situazione è definita come **eccezione imprecisa**. In questo caso dovrò gestire l'eccezione della SUB e per non perdere il lavoro delle istruzioni precedenti dovrò salvare i risultati della DIV e della ADD bufferizzandole. Potrei cercare di accorgermi prima delle eccezioni aritmetiche, già nella fase di IDecode.

## R4000

È un processore a 64bit introdotto nel 1991.

R4000 è caratterizzato da un'architettura a pipeline composta da 8 stadi.

La lunghezza di questa pipeline comporta:

- Più interconnessioni per effettuare il forwarding
- Cresce il load delay slot (2 cicli)
- Cresce il branch delay slot (3 cicli)

### Load Delay Slot

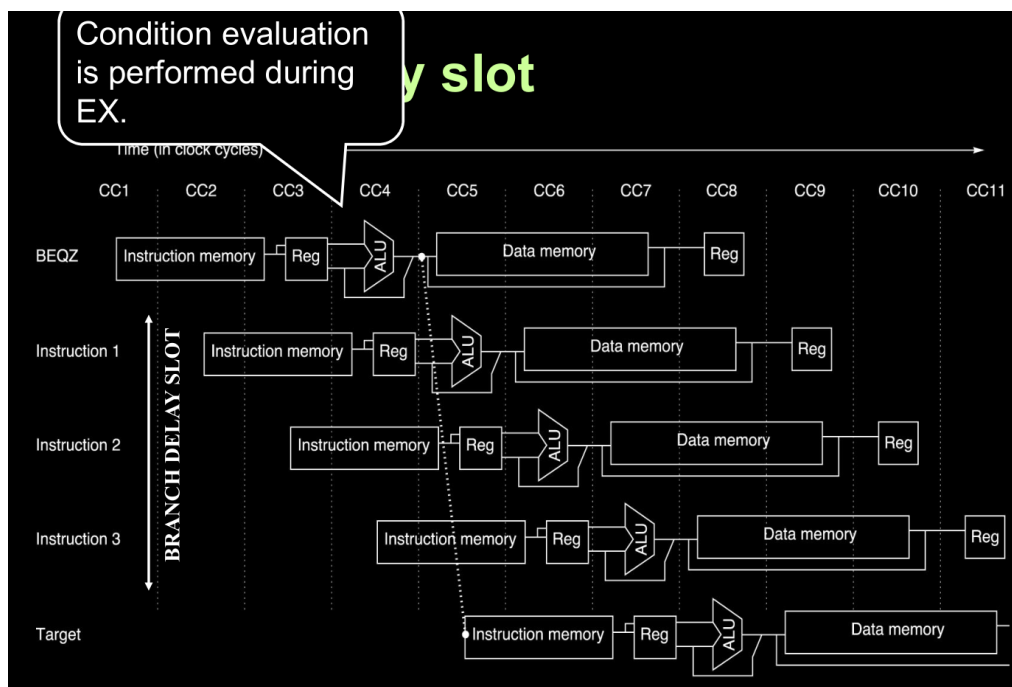
Un'istruzione di load recupererà l'operando dalla memoria nello stadio di MEM e questo sarà disponibile in questo stadio grazie all'operazione di forwarding.

L'operando non sarà dunque disponibile immediatamente.

Un'istruzione che vorrà utilizzare quell'operando dovrà attendere dei cicli di clock prima di poterlo utilizzare.

Lo spazio fra un'istruzione di load e l'istruzione che adopera l'operando della load è denominato Load Delay Slot.

Il compilatore può sfruttare il load delay slot inserendo delle istruzioni che non adoperano l'operando della load in questo spazio.



Load Delay Slot, è il numero di istruzioni (o stallo) che devo inserire per permettere di evitare un hazard di dato. Se la pipeline è più lunga dovrò allora inserire più stalli per gestire queste dipendenze di dato.

Per evitare questi stalli posso sfruttare questi spazi per inserire delle istruzioni precedenti ad un'istruzione tipo load in modo da riempire il load delay slot. Basta che non cambi il funzionamento del programma.

L'assemblatore è implementato in modo da cercare delle istruzioni che mi permettano di riempire il load delay slot in modo da ottimizzare i colpi di clock, evitando gli stalli.

## Branch Delay Slot

Tale operazione può essere svolta anche per i salti -> Branch Delay Slot.

**Il Branch Delay Slot è lo spazio d'istruzioni che posso sfruttare dopo un'istruzione di salto.**

Nell'architettura dei microprocessori la **branch delay instruction** è l'istruzione immediatamente successiva ad una condizione di salto condizionato che viene eseguita indipendentemente dal fatto che il salto vada eseguito oppure no.

Vogliamo spostare delle istruzioni al posto di inserire stalli quando incontriamo un'istruzione di branch, in caso di dipendenze di dato oppure in caso di scritture concorrenti nella stesso tempo di clock.

*/\*Per le istruzioni floating point posso inserire delle istruzioni in caso di dipendenza di dato.*

*Nell'esempio la add va in stallo perché ha una dipendenza di dato di F3 dalla mul, posso quindi spostare da daddi R2, R2, 8 fra la div e la add, eliminando così la dipendenza di dato. Risparmiando così 1 colpo di clock.\**

## **Parallelismo – instruction level parallelism - ilp**

Utilizziamo la pipeline per sfruttare il parallelismo hardware.

Dobbiamo quindi cercare delle istruzioni che non hanno dipendenze di dato che possono essere risolte in parallelo sfruttando la pipeline.

Abbiamo 2 approcci per la ricerca delle istruzioni indipendenti:

- Statico: Il compilatore, conoscendo la pipeline, cerca l'ordine ottimale delle istruzioni che minimizza il numero di stalli, quindi il numero di clock per eseguire il codice.  
**Analizza il codice, conosce la pipeline, definisce l'ordine.**  
Più facile da implementare, processori con HW più semplice.  
Questa soluzione si preferisce nel mondo dei sistemi embedded.
- Dinamico: Il compilatore genera solo le istruzioni assembler necessarie. L'ordine di queste viene deciso runtime dal processore. Sarà dunque il processore a garantire l'ordine migliore.  
L'intelligenza la sposto dal compilatore al processore. (processori superscalari)  
In questo modo massimizzo la portabilità del codice, perché il processore schedula al meglio il codice per sé stesso.

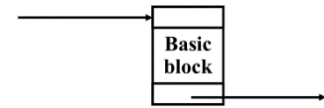


## Approccio statico

Le istruzioni generate sono **ordinate nei basic block** (blocchi fra 2 salti).

All'interno del basic block sono certo che le istruzioni le eseguirò in sequenza senza salti. Posso quindi spostare queste istruzioni stando attento solo alle dipendenze di dato. I compilatori cercano principalmente un ordinamento ottimale delle istruzioni dentro i basic block.

No branches in, except to the entry  
No branches out, except at the exit.



Nell'esempio a causa di vincoli di dipendenza di dato (siccome LD carica realmente il dato nella fase di mem) devo inserire degli stalli. Con la nuova riorganizzazione non abbiamo nessuno stallo ottenendo 12 colpi di clock (il minimo)

LD	Rb, b	IF	ID	EX	MEM	WB				5			
LD	Rc, c		IF	ID	EX	MEM	WB			1			
ADD	Ra, Rb, Rc		IF	ID	st	EX	MEM	WB		2			
SD	Ra, Va		IF	st	ID	EX	MEM	WB		1			
LD	Re, e			IF	ID	EX	MEM	WB		1			
LD	Rf, f				IF	ID	EX	MEM	WB	1			
SUB	Rd, Re, Rf					IF	ID	st	EX	MEM	WB	2	
SD	Rd, Vd						IF	st	ID	EX	MEM	WB	1
<hr/>													
LD	Rb, b	IF	ID	EX	MEM	WB							
LD	Rc, c		IF	ID	EX	MEM	WB						
LD	Re, e		IF	ID	EX	MEM	WB						
ADD	Ra, Rb, Rc		IF	ID	EX	MEM	WB						
LD	Rf, f			IF	ID	EX	MEM	WB					
SD	Ra, Va				IF	ID	EX	MEM	WB				
SUB	Rd, Re, Rf					IF	ID	EX	MEM	WB			
SD	Rd, Vd						IF	ID	EX	MEM	WB		

Quando parliamo di un basic block dobbiamo considerare che più la dimensione del **basic block grande** allora più saranno i gradi di libertà per effettuare un'ottimizzazione da parte del compilatore.

Solitamente però i basic block hanno dimensioni piccole, **vogliamo quindi aumentare le dimensioni del basic block.**

Abbiamo diverse tecniche, svolte dal compilatore:

- Giochiamo sui cicli, ad esempio **possiamo fare un lavoro di loop unrolling** cercando di diminuire il numero di volte in cui ciclo viene eseguito ma aumentando il corpo del ciclo riportando più istruzioni nello stesso ciclo. Cerchiamo dunque di diminuire i salti aumentando anche la dimensione del basic block. **Abbiamo quindi un doppio vantaggio -> meno salti e basic block più grande.** La **dimensione del codice sarà però maggiore** e nel caso dei **sistemi embedded** questo è un **problema perché la memoria è dimensionata sulla dimensione esatta del codice eseguibile.**
- Si può anche fare un miglioramento di cui parleremo successivamente

for (i=0; i<N; i++)	for (i=0; i<N/4; i++)
{	{
body	body
}	body
	body
	body
	}

All'interno del basic block, il compilatore deve effettuare l'analisi delle dipendenze in modo da poter ottimizzare l'ordine delle istruzioni.

## Tipi di dipendenze

Due istruzioni sono indipendenti se possono essere eseguite in parallelo senza stalli.

Se due istruzioni sono dipendenti devo eseguirle in ordine.

- **Di dato:** due istruzioni in sequenza usano lo stesso dato. Anche il risultato scritto in memoria e poi usato subito da un'istruzione successiva. Se faccio operazioni di add e subito sub posso eliminare la dipendenza con il data forwarding. In caso invece di load e poi add ho una dipendenza di dato. La pipeline mi condiziona quindi le dipendenze di dato. Non tutte le dipendenze quindi creano un hazard.

Devo quindi considerare le dipendenze e gli hazard. Hazard lower bound di dipendenza.

Gli hazard sono diversi in base alla pipeline dove viene piazzato lo stesso codice. Al pari numero di dipendenze.

Le dipendenze di dato fra registro o dato in memoria sono differenti da identificare. Quelle fra registri sono più facili. ST R5, 30(R8) LD R4, 10(R2) posso avere una dipendenza di dato in base al valore di r8 ed r2 perché posso accedere alla stessa cella di memoria. I compilatori in questo caso si mettono nel caso più conservativo e non invertono le operazioni perché non possono identificare, a tempo di compilazione, se c'è una dipendenza di dato a causa di un accesso alla stessa cella di memoria.

- **Di controllo:** sono quelle legate ai salti
- **Di nome:** sono delle dipendenze associate a coppie di istruzioni che usano lo stesso registro dove però non c'è un flusso di dati fra di esse.

- Vi sono delle dipendenze chiamate antidipendenze che prevedono un utilizzo dei dati comune anche se non sono operazioni subito successive. In particolare, abbiamo un'antidipendenza quando, avendo due istruzioni *i* e *j*, l'istruzione *j* scrive nello stesso registro dove l'istruzione *i* legge.

Ad esempio:

```
LD R1, 0(R3)
```

```
ADD R1, R2, R4
```

questa è un'antidipendenza su R1.

### Example

```
Loop:  L.D    F0, 0(R1)
        ADD.D F4, F0, F2
        S.D    F4, 0(R1)
        L.D    F0, -8(R1)
        ADD.D  F4, F0, F2
        S.D    F4, -8(R1)
        L.D    F0, -16(R1)
```

Antidependence

- Vi è anche una dipendenza di output dependence. Due istruzioni in ordine scrivono sullo stesso registro. In questo caso non posso scambiare l'ordine di queste operazioni perché anche se sono istruzioni che operano per parti di codice differente possono fare accesso alla stessa risorsa. Le Name Dependencing possono essere risolte con un meccanismo chiamato register renaming.

### Example

```
Loop:  L.D    F0, 0(R1)
        ADD.D  F4, F0, F2
        S.D    F4, 0(R1)
        L.D    F0, -8(R1)
        ADD.D  F4, F0, F2
        S.D    F4, -8(R1)
        L.D    F0, -16(R1)
```

Output  
dependence

In caso di antidependence il compilatore la identifica e cambia il registro che crea la dipendenza con un altro registro. Per output dependence posso utilizzare lo stesso concetto di register renaming.

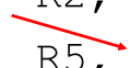
## Hazard e data dependencies

Gli Hazard visti sono:

- RAW (read after write) data dependencies
- WAW (write after write) name dependencies
- WAR sono il risultato delle dipendenze di tipo antidependence. Non posso cambiare l'ordine delle istruzioni se una legge e una scrive utilizzando lo stesso dato letto. Name dependencies
- RAR non produce mai hazard.

### RAW

DADD R1, R2, R3  
DSUB R4, R5, R1



### WAW

LW R1, 0(R2)  
DADD R1, R2, R3

## Dipendenze di controllo

La dipendenza di controllo è introdotta dai salti.

**Basic block -> porzione di codice contigua priva di salti. Siamo sicuri che queste istruzioni li eseguiremo in sequenza.**

In caso di salti "annidati" avrò dei basic block più frammentati, le dipendenze di controllo impediscono il trasferimento delle istruzioni da un basic block ad un altro.

Sotto certe condizioni, che vedremo fra poco, possiamo però violare i vincoli di controllo.

## Dipendenze eccezioni

Bisogna rispettare anche le dipendenze delle eccezioni, dobbiamo quindi attenzionare l'eventuale nascita di eccezioni in caso di spostamento delle istruzioni.

## Data flow

L'ordine dei dati influenza il risultato finale delle istruzioni.

Grafo dipendenze controllo, se gli interessa violare le dipendenze deve verificare che a valle della modifica che i risultati prodotti siano consistenti

In un approccio conservativo non ci conviene quindi spostare le istruzioni attraverso istruzioni di salto condizionato.

Alcuni spostamento d'istruzioni sono però consentiti, anche oltre il vincolo delle dipendenze di controllo. Spostando così istruzioni da un basic block ad un altro.

**Riassunto: Vogliamo spostare le istruzioni attraverso i basic block per aumentare il parallelismo ma dobbiamo stare attenti alle dipendenze di dato poiché l'ordine delle istruzioni influenza il comportamento del codice. Vi sono delle tecniche per aumentare l'efficienza del processore in presenza di istruzioni di salto.**

## Predizione dei Salti

I salti riducono il CPI (cicli per istruzione - efficienza della pipeline). In alcuni caso possiamo effettuare una previsione su un salto in modo da influenzare il funzionamento della pipeline per le istruzioni successive. Le istruzioni caricate dalla pipeline saranno quindi influenzate dalle previsioni effettuate sui salti condizionati

## Come effettuare le previsioni di salto

Tecniche:

- Statiche: fatte dal compilatore, calcola la probabilità che un salto venga preso o non preso. Vengono fatte a priori, prima dell'esecuzione del codice. Una volta che la previsione è presa non può essere cambiata in runtime.
- Dinamiche: Vengono fatte al tempo d'esecuzione dal processore. La previsione può cambiare nel tempo a runtime.

## Tecniche statiche

Ci sono tre grandi famiglie all'interno delle tecniche statiche:

- Always: considera i salti sempre presi (tecnica stupida)
- Consideriamo i salti presi o non presi se sono all'indietro o in avanti. Tipicamente i compilatori prendono i salti come presi se saltano all'indietro e come non presi se i salti sono in avanti. Il salto viene classificato come non preso se è in avanti.
- Eseguire una o più volte il codice e per simulazione vado a contare quante volte un salto è preso o non preso. Uso questa statistica come previsione per l'esecuzione del codice. L'efficacia di questa modalità dipende fortemente dai dati di stimolo in ingresso (che devono corrispondere ai dati che poi realmente gireranno sul codice) e dal numero di predizioni.

L'informazione deve essere passata al processore. Alcuni processori forniscono dalle istruzioni al compilatore in base alla previsione del salto preso o del salto non preso. Tale previsione permette al processore di caricare in pipeline istruzioni in base alla previsione del salto condizionato.

## Tecniche dinamiche

Eseguite dal processore. L'hw deve essere più elaborato. Le tecniche usate sono:

- **Branch history table (BHT):** il processore può ricordarsi per ciascun salto condizionato cosa aveva eseguito la volta precedentemente. Nella fase di startup il processore non sarà in grado di produrre una previsione ma per le volte successive farà un'istruzione di salto sarà condizionata dal suo comportamento al passo precedente.

Ciò che aveva fatto la volta precedente con quel salto sarà la previsione per la prossima volta che il processore incontrerà quell'istruzione di salto condizionato.

Questa tecnica richiede l'esistenza di una tabella in HW che contenga tutte le istruzioni di salto condizionato e gli esiti precedenti. Tale tecnica è però non implementabile in questo modo perché non possiamo avere sempre tutto lo spazio disponibile. (visto che il numero di salti in un codice è variabile).

Applichiamo quindi un'ulteriore approssimazione bloccando la dimensione massima della tabella delle corrispondenze.

Il processore guarda l'indirizzo dell'istruzione di salto condizionato, prende i bit bassi che selezionano la parola della tabella che contiene la previsione del salto.

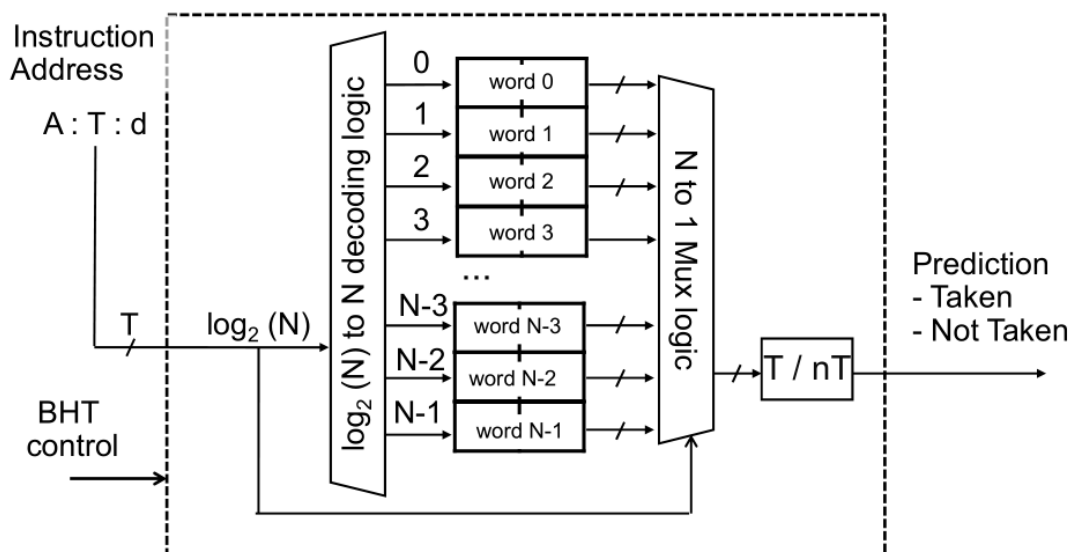
È una funzione di mapping molto semplice che però lascia spazio ad errori nel caso due istruzioni di salto abbiano gli stessi bit bassi dell'indirizzo. Questa tecnica però è facile e poco costosa da implementare.

Allo start del processore possiamo avere:

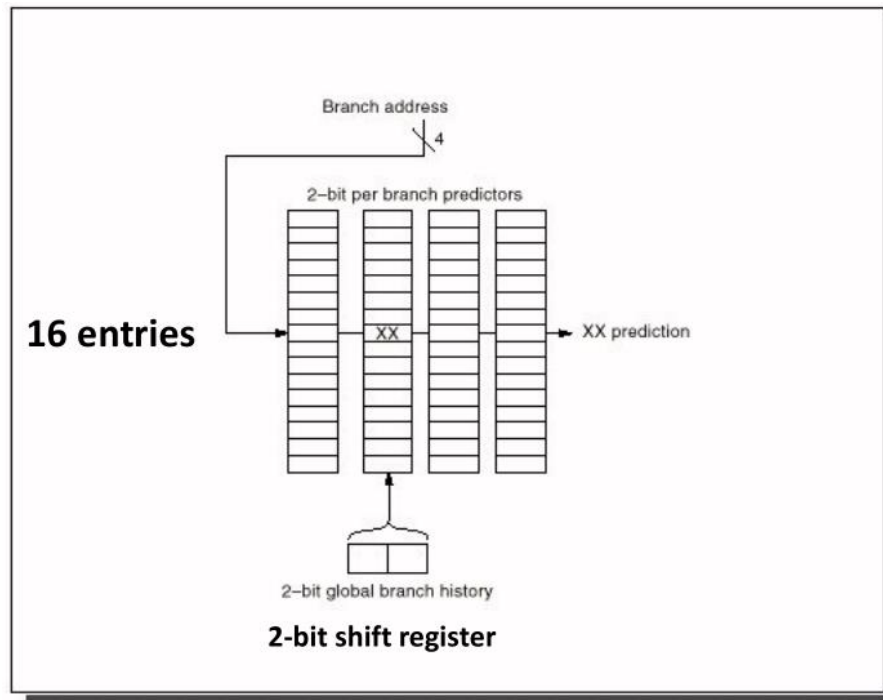
- Tutti 0, considero tutti i salti untaken
- Posso inizializzare la tabella con una previsione statica

Questa tecnica può essere sviluppata utilizzando due bit, invece che uno, nella tabella della previsione. Questi due bit possono assumere 4 combinazioni che indicano se una previsione è presa frequentemente, presa non frequentemente, non presa non frequentemente, non presa frequentemente. È un contatore di 2 bit che parte da 00 (frequentemente non preso) e ogni volta che eseguo il salto aggravo il contatore facendo un +1. Tale tecnica funziona meglio rispetto al predittore a 1bit.

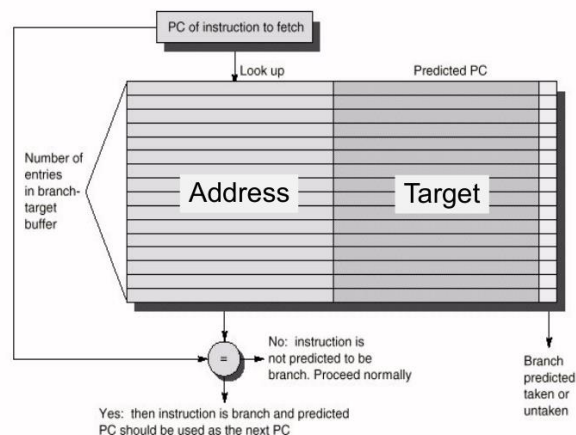
Il funzionamento di questo metodo di previsione è influenzato dalla dimensione della tabella. Spesso la tabella si dimensiona sulla lunghezza del codice (più semplice per sviluppo embedded)



- Correlating Predictors (predittori correlati):** Ad una data istruzione di salto condizionato posso arrivare da cammini diversi. Osserviamo l'esito dei precedenti salti condizionati eseguiti prima di quello di cui voglio fare la predizione. La predizione è basata non più su cosa aveva fatto quel salto nel passato ma vado a vedere gli ultimi (tipicamente 2) salti prima del salto interessato e vado a vedere cosa era successo in passato al mio salto arrivando dai precedenti salti. Posso generare 4 possibili scenari (essendoci 2 salti precedenti). Avrò quindi 4 tabelle (BHT), ognuna di 2bit, che tengono traccia della storia precedente dei salti precedenti al salto interessato. Ho uno shift register di due bit che tiene traccia della storia degli ultimi due salti. Unisco i bit dello shift register ai 10 bit per indirizzare una BHT in modo da poter indirizzare 4 BHT, ognuna da  $2^{10}$  (1024) righe. Userò quindi un indirizzo di 12 bit ->  $2\text{shift} + 10\text{parte bassa}$



## Branch-Target Buffer



Vogliamo predire non solo l'esito del salto (Branch History Table) ma anche l'indirizzo a cui salteremo. Prediremo non solo le istruzioni di salto condizionato ma anche quelle di salto incondizionato.

Per accedere al Branch-Target Buffer uso lo stesso meccanismo della Branch History table. Accedo alla Branch-Target Buffer accedendo con i bit meno significativi di un'istruzione. La differenza è che accedo a questa tabella prima ancora di capire se è un'istruzione di salto o meno.

La dimensione della Branch-Target buffer è una potenza di 2 ->  $2^k$

Prendo quindi i k bit meno significativi dell'istruzione per accedere al branch-target buffer che contiene l'indirizzo completo dell'istruzione di cui posso fare il fetch.

L'elemento può contenere o meno un'informazione relativa all'istruzione di cui sto facendo il fetch. Dentro quella cella ho l'indirizzo dell'istruzione che sto caricando, se ho una corrispondenza allora guardo il secondo campo che contiene l'indirizzo a cui saltare.

Il Branch Target Buffer ha quindi 2 campi:

- Indirizzo dell'istruzione di salto
- Indirizzo a cui saltare

Se ho corrispondenza fra l'indirizzo dell'istruzione di salto dell'istruzione che devo eseguire e l'indirizzo dell'istruzione di salto nel primo campo del branch-target buffer, eseguo un salto utilizzando l'indirizzo contenuto nella linea corrispondente nella branch target buffer.

L'istruzione di cui posso fare il fetch può essere:

- Salto incondizionato: non ho problemi nel saltare in caso di match. Carico il fetch dell'indirizzo a cui saltare subito dopo l'istruzione del salto.
- Salto Condizionato: In questo caso dovrò valutare se la predizione è vera o falsa (taken o non taken)
- Istruzione non di salto: Accedo sempre al branch-target buffer ma non farò mai match con l'indirizzo dell'istruzione e le istruzioni nel branch-target buffer. In questo caso non farò quindi nessun salto.

Il campo del bit che indica la predizione del salto non viene usato.  
GUARDA IL FLOW CHART del branch-target Buffer nelle slides.

La scrittura dentro la branch-target buffer viene effettuata quando effettuo un salto.

Il confronto dell'address avviene fra l'indirizzo nel PC (tutto) e la parte contenuta nel campo address.

## Dynamic Scheduling Techniques – Approccio Dinamico

Nello scheduling dinamico il processore sceglie in hardware l'ordine d'esecuzione delle istruzioni. In questo modo possiamo massimizzare l'uso delle risorse hardware e quindi velocizzare l'esecuzione.

L'esecuzione dell'istruzione avviene in ordine differente rispetto a come compaiono nel codice in base al fatto se i blocchi di calcolo della specifica operazione sono disponibili oppure no.

Ad esempio, se l'unità funzionale che effettua la SUB è disponibile, posso far processare un'operazione di sub prima delle altre.

In questo caso devo fare attenzione alle dipendenze di dato di tipo RAW oppure WAW e WAR.

```
DIV F0, F2, F4
ADD F6, F0, F8
SUB F8, F9, F14
MUL F6, F10, F8
```

Se eseguo prima la MUL prima della ADD poi alla fine in F6 potrei ritrovare il valore della ADD e non della MUL.

Vogliamo quindi avere un'esecuzione del codice di tipo out of order in modo da migliorare le performance del nostro processore.

### Fase di Decoding

Per l'esecuzione out of order devo dividere in due parti lo stadio di decodifica. Bisogna quindi dividere la decodifica e l'accesso agli operandi. Questo perché devo capire per ogni istruzione quale blocco di istruzione devo usare.

- **Issue:** assegno l'operazione all'unità funzionale adeguata. Devo anche capire se l'unità in questione è libera oppure sta ancora lavorando. In questa parte avviene il riconoscimento delle dipendenze funzionali fra le istruzioni.
- **Read Operands:** serve per accedere ai registri che verranno usati per l'esecuzione dell'istruzione. Tale operazione sarà anche addetta al riconoscimento degli Hazard. Deve quindi riconoscere se un dato operando è in uso oppure no.

Se un'unità funzionale è occupata allora l'esecuzione di una data istruzione dovrà attendere. (Issue)

### Fase di Execute

Ci saranno più stadi di execute, una per ogni operazione che posso eseguire (ADD, SUB, DIV etc...)



## Hardware schemes for dynamic scheduling - TOMASULO

L'ingegnere Tomasulo della IBM inventò negli anni 60 un tipo d'architettura.

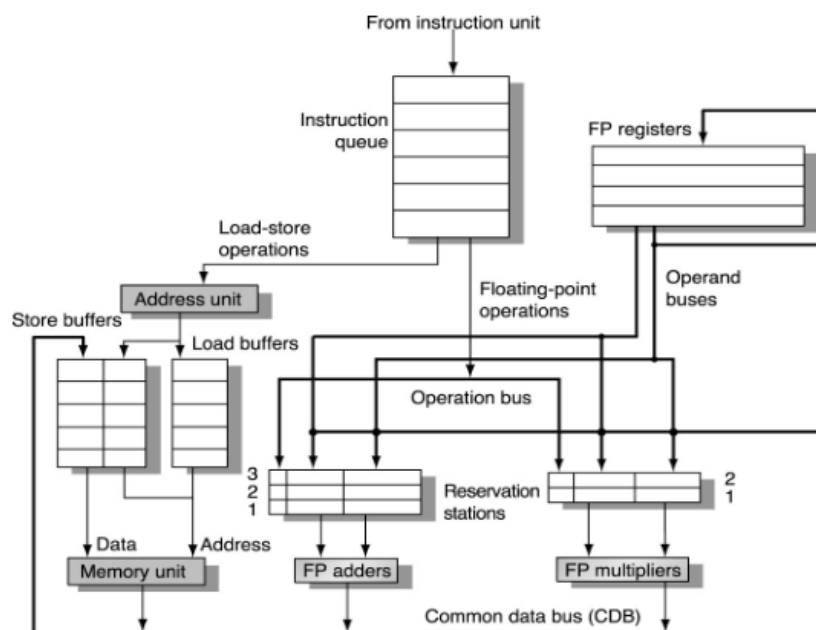
Per ogni unità funzionale ci sono un certo numero di reservation station. Sono sostanzialmente dei buffer in cui si accumulano gli operandi delle istruzioni che dovranno essere usati dalle unità di calcolo (somma, differenza etc..)

Quando un'istruzione viene assegnata ad un'unità funzionale (fase di ISSUE) si assegna all'istruzione uno slot della reservation station dove vi sono gli operandi. **Dentro la reservation station vi sono, oltre alle informazioni sulle istruzioni e sugli operandi, le informazioni sull'istruzione che sta utilizzando l'operando se c'è da aspettare.** Finché tutti e due gli operandi non sono disponibili si aspetterà. Gli operandi vengono trasferiti fra tutte le reservation station tramite un bus denominato Common Data Bus. (CDB)

Nella reservation station vi è scritto:

- Operazione da eseguire
- Operandi
- Se un operando non è disponibile vi è scritto quale istruzione deve produrre l'operando che serve

**Le reservation station hanno visione del common data bus**, in caso di produzione dell'operando interessato a parte di una reservation station, questo verrà trasferito attraverso il common data bus e tutte le reservation station che vorrebbero prendersi l'operando potranno prenderselo.



### Unità di Load e Store

1. Deve calcolare l'indirizzo con cui accedere in memoria
2. accedere in memoria

Con l'architettura di Tomasulo si possono formare delle dipendenze di dato poiché load e store possono essere scambiate durante l'esecuzione poiché non conosciamo il valore del registro di cui devo fare load e store. Sulle architetture viste fino ad ora, i compilatori, non invertono mai una load e una store.

Tale limitazione viene superata nell'architettura di Tomasulo cercando di capire se vi sono delle dipendenze di dato ed eventualmente bloccare o meno tali istruzioni.

Dentro l'unità di load e store vi è un buffer in cui sono memorizzate le operazioni di load e store in attesa di completamento.

Se vi è un'istruzione di load devo vedere se vi è un'istruzione di store che lavora sullo stesso indirizzo.

In caso di presenza di una store che carica nello stesso indirizzo di una load allora attenderò il completamento della store. Non potendo così invertire le operazioni di load e store rispetto all'ordine presente nel codice.

Se all'unità di issue arriva un'istruzione di salto dobbiamo bloccare l'esecuzione di tutte le istruzioni successive.

Tale meccanismo rende complicato l'abort di un'istruzione di salto se questo non era stato predetto correttamente.

Quindi quando arriva un salto dobbiamo essere certi che tutte le istruzioni successive non vengano eseguite fintanto che non sappiamo la condizione del salto. Questo problema viene gestito da un meccanismo chiamato ***speculation***.

Le unità funzioni scrivono dentro il register file attraverso il common data bus. Il common data bus implementa il data forwarding.