

# Appunti di Architetture Avanzate dei Calcolatori

Matteo Gianello

7 maggio 2014

Quest'opera è stata rilasciata con licenza Creative Commons Attribuzione - Non commerciale - Condividi allo stesso modo 3.0 Unported. Per leggere una copia della licenza visita il sito web <http://creativecommons.org/licenses/by-nc-sa/3.0/deed.it> .

# Indice

<b>1</b>	<b>Pipelining</b>	<b>3</b>
1.1	Concetti base . . . . .	3
1.1.1	Reduced Instruction Set nei processori MIPS . . . . .	3
1.1.2	Esecuzione delle istruzioni . . . . .	6
1.1.3	Implementazione base di un MIPS . . . . .	7
1.2	Pipelining . . . . .	10
1.2.1	Implementazione di una Pipeline . . . . .	12
1.3	Il problema del "Hazard" . . . . .	15
1.3.1	Data Hazard . . . . .	15
1.3.2	Altri tipi di Data Hazard . . . . .	17
1.4	Analisi delle performance . . . . .	18
<b>2</b>	<b>Tecniche di predizione dei salti</b>	<b>22</b>
2.1	Il problema del Control Hazard . . . . .	22
2.2	Tecniche di predizione dei salti . . . . .	23
2.2.1	Tecniche di predizione statiche . . . . .	25
2.2.2	Tecniche di predizione dinamiche . . . . .	27
2.3	Speculazione . . . . .	31
<b>3</b>	<b>Instruction Level Parallelism</b>	<b>34</b>
3.1	Tipi di Hazards sui dati . . . . .	34
3.2	Parallelismo a livello di istruzione . . . . .	35
3.2.1	ILP in pratica . . . . .	37
3.2.2	Esecuzione super-scalare e VLIW . . . . .	38
3.3	Scoreboard . . . . .	39
3.4	Algoritmo di Tomasulo . . . . .	42
3.4.1	Gli stadi dell'algoritmo di Tomasulo . . . . .	43
3.4.2	Alcuni dettagli . . . . .	44
3.4.3	Tomasulo in pratica . . . . .	44
3.4.4	Tomasulo vs Scoreboard . . . . .	44
3.5	Register Renaming . . . . .	45
3.5.1	Renaming implicito . . . . .	45
3.5.2	Renaming esplicito . . . . .	45

# Introduzione

Il corso di Architetture Avanzate dei Calcolatori si prefigge lo scopo di fornire una vista sulle più recenti architetture avanzate dei calcolatori, introducendo i meccanismi base delle micro-architetture che si possono ritrovare nei moderni microprocessori, ed infine fornire le ragioni dietro alle tecniche adottate nelle architetture dei computer.

## 1 Pipelining

### 1.1 Concetti base

Definiamo prima di tutto quali sono le principali caratteristiche dell'architettura MIPS partendo dalla definizione delle istruzioni utilizzate da questi calcolatori. Esistono due tipi di istruzioni le CISC e le RISC; le istruzioni di tipo **CISC** (*Complex Instruction Set Computer*) sono un set di istruzioni esteso che permettono ai processori di eseguire operazioni molto complesse come somme tra operandi caricati direttamente dalla memoria centrale, le istruzioni **RISC** (*Reduced Instruction Set Computer*), invece, sono istruzioni semplici che possono essere eseguite in un unico ciclo di clock e ottimizzate per le performance sulle CPU CISC. Le architetture RISC sono anche dette architetture di tipo *LOAD/STORE*, in quanto le istruzioni non accedono direttamente ai dati in memoria ma accedono ai dati contenuti in registri del processore, solo due istruzioni permettono l'accesso alla memoria principale, queste due istruzioni sono:

- **load** che carica i dati dalla memoria ai registri.
- **store** che sposta i dati dai registri alla memoria.

Un'altra caratteristica fondamentale per le architetture RISC è l'utilizzo della *Pipeline* una tecnica di ottimizzazione basata sull'esecuzione sovrapposta di molteplici istruzioni sequenziali.

#### 1.1.1 Reduced Instruction Set nei processori MIPS

Vediamo ora quali sono le diverse istruzioni di tipo RISC e come sono rappresentate nel calcolatore

**Istruzioni ALU** Vediamo innanzitutto le istruzioni di somma ovvero quelle eseguite dalla ALU. Queste possono essere di due tipi, Una istruzione di tipo *R-Format* è un'istruzione che prende in considerazione due registri, tale tipo di operazione è applicabile solo alle istruzioni **add** di tipo registro-registro. Esistono poi le **addi** che viene chiamata anche somma *immediata* in quanto avviene tra un registro ed un valore costante. Tale tipo di istruzione è del tipo *I-Format*. Lo pseudo codice assembly delle due istruzioni è mostrato qui di seguito, inoltre possiamo anche vedere come vengono svolte le operazioni.

```
add  $s1, $s2, $s3      # $s1 <- $s2 + $s3
addi $s1, $s2, 4         # $s1 <- $s2 + 4
```

In Figura 1 vediamo come è suddivisa un'istruzione di tipo *R-Format* I diversi campi indicano rispettivamente:

**op** identifica il tipo di istruzione ALU da eseguire

**rs** indica il registro nel quale è contenuto il primo operando

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

Figura 1: Esempio di istruzione ALU di tipo R-Format

**rt** indica il registro nel quale è contenuto il secondo operando

**rd** indica il registro di destinazione

**shamt** sta ad indicare i bit di shift amount

**funct** identifica i diversi tipi di istruzione

op	rs	rt	immediate
6 bit	5 bit	5 bit	16 bit

Figura 2: Divisione delle informazioni in un registro di un istruzione ALU di tipo diretto

Nella Figura 2 vediamo invece la suddivisione di un registro nel caso di un operazione di ALU immediata. La suddivisione dei diversi campi è la seguente:

**op** identifica l'istruzione di tipo immediato

**rs** indica il registro nel quale è posizionato il primo operando

**rt** indica il registro di destinazione del risultato

**immediate** contiene il valore per l'operazione immediata nel range  $-2^{15}$  e  $+2^{15} - 1$

**Istruzioni LOAD/STORE** Le istruzioni di *load* e di *store* sono quelle che permettono di caricare e scaricare i valori dai registri della CPU alla memoria centrale e viceversa. Un esempio di codice assembly per le istruzioni load e store.

```
lw $s1, offset($s2) # $s1 <- M[$s2 + offset]
sw $s1, offset($s2) # M[$s2 + offset] <- $s1
```

In Figura 3 vediamo come sono strutturate le istruzioni di *load* e di *store*; come possiamo vedere anche queste istruzioni sono nel formato *I-Format*

La suddivisione del registro è la seguente:

op	rs	rt	offset
6 bit	5 bit	5 bit	16 bit

Figura 3: Struttura di un'istruzione tipo load/store

**op** identifica l'istruzione di tipo load o store

**rs** identifica il registro base

**rt** identifica il registro sorgente o destinazione per i dati delle operazioni di store o di load da o per la memoria

**offset** da sommare all'indirizzo contenuto in *rs* per calcolare l'indirizzo di memoria

**Istruzioni di salto** Per quanto riguarda le istruzioni di salto possiamo suddividerle in due categorie, i salti *condizionati*, che richiedono la verifica di una determinata condizione per decidere se effettuare un salto; oppure istruzioni di salto *incondizionato* che effettuano il salto sempre. Lo pseudo assembly di un'istruzione di salto condizionato è:

```
beq $s1, $s2, L1 #go to L1 if ($s1 == $s2)
bne $s1, $s2, L1 #go to L1 if ($s1 != $s2)
```

Le istruzioni di salto condizionato sono nel formato *I-Format*. La suddivisione del registro per questo tipo di istruzione è mostrata in Figura 4

op	rs	rt	address
6 bit	5 bit	5 bit	16 bit

Figura 4: Struttura di un'istruzione tipo branch condizionato

**op** identifica l'istruzione di tipo branch condizionale

**rs** identifica il primo registro da comparare

**rd** identifica il secondo registro da comparare

**address** identifica l'offset rispetto al PC che corrisponde all'indirizzo dell'etichetta

Per quanto riguarda il salto incondizionato il funzionamento è molto più semplice, quando si raggiunge l'istruzione di salto il PC punta direttamente all'istruzione indicata dall'etichetta. Tale semplicità si rispecchia nella struttura dell'istruzione che in questo caso è di tipo *J-Format*. Lo pseudocodice assembly dell'istruzione di salto è:

```
j L1 #go to L1
jr $s1 #go to add. contenuto in $1
```

Un esempio di struttura di istruzione di salto è rappresentato in Figura 5 dove i valori indicano

op	address
6 bit	26 bit

Figura 5: Divisione delle informazioni in un registro di un'istruzione tipo branch incondizionato  
rispettivamente

**op** identifica il tipo di istruzione

**address** identifica l'indirizzo della prossima istruzione da eseguire

Ricapitolando possiamo suddividere le istruzioni in tre categorie in base alla loro struttura e a come vengono eseguite:

- Tipo R (*Registro*)
  - Istruzione ALU
- Tipo I (*Immediate*)
  - ALU immediate
  - Istruzioni Load/Store
  - Istruzioni di salto condizionato
- Tipo J (*Jump*)
  - Istruzioni di salto incondizionato

Il perché di questa divisione lo si capisce molto facilmente dallo schema in Figura 6 nel quale vengono confrontati le diverse suddivisioni degli Instruction Register.

	6-bit		5-bit		5-bit		5-bit		5-bit		6-bit	
	31	26	25	21	20	16	15	11	10	6	5	0
R	op		rs		rt		rd		shamt		funct	
I	op		rs		rt		offset/immediate					
J	op		address									

Figura 6: Divisione dei registri nei diversi casi di operazione

### 1.1.2 Esecuzione delle istruzioni

Vediamo ora come possono essere implementate le diverse istruzioni in ambiente MIPS. Tutte le istruzioni possono essere implementate suddividendo l'esecuzione in cinque fasi distinte:

1. **Instruction Fetch Cycle:** durante questo ciclo viene inviato il contenuto del *Program Counter* all'*Instruction Memory* e viene prelevata l'istruzione corrispondente. Successivamente viene aggiornato il PC perchè punti alla prossima istruzione aggiungendo 4 al valore attuale (le istruzioni sono di 4 bytes)
2. **Instruction Decode and Register Read Cycle:** in questo ciclo si decodifica l'istruzione corrente e si leggono dal *Register File* i registri necessari corrispondenti ai registri specificati nei campi dell'istruzione. Si fa inoltre l'estensione del segno nel caso sia necessario.
3. **Execution Cycle:** In questo ciclo la ALU effettua le operazioni sugli elementi che sono stati preparati nel ciclo precedente. In base alle istruzioni la ALU esegue le seguenti operazioni
  - Istruzione ALU registro-registro: la ALU esegue le operazioni sugli operandi che ha letto dal *Register File*

- Istruzioni ALU immediate: la ALU esegue l'operazione specificate sul primo operando letto dal *Register File* e sul operando immediato al quale è stata applicata l'estensione di segno.
  - Istruzioni Load/Store la ALU aggiunge all'indirizzo base l'offset per calcolare l'indirizzo effettivo.
  - Istruzioni di salto condizionato: la ALU compara i due registri e calcola l'indirizzo target del salto da aggiungere al PC
4. **Memory Access (ME):** durante questo ciclo le istruzioni di *Load* effettuano la lettura dalla memoria usando l'indirizzo effettivo calcolato al ciclo precedente, le istruzioni di *Store* scrivono nella memoria i dati provenienti dal registro, infine, le istruzioni di *Branch* aggiornano il valore del Program Counter con l'indirizzo target calcolato al passo precedente, nel caso in cui la condizione sia verificata.
  5. **Write-Back Cycle (WB):** in questo ciclo le istruzioni di *Load* scrivono i dati letti dalla memoria nel registro di destinazione mentre le istruzioni di *ALU* scrivono il risultato delle operazioni nei registri di destinazione.

Come possiamo notare le diverse operazioni non usano sempre tutti i cicli appena descritti ma solitamente (tranne nel caso della *load*) attraversano solo alcune fasi come possiamo vedere dallo schema in Figura 7.

Mentre nella tabella di Figura 8 possiamo vedere le latenze di ogni operazione nel caso di tempo di ciclo uguale ad 1 *ns*.

**ALU Instructions: `op $x, $y, $z`**

Instr. Fetch & PC Increm.	Read of Source Regs. \$y and \$z	ALU OP ( $y \text{ op } z$ )	Write Back of Destinat. Reg. \$x
------------------------------	-------------------------------------	---------------------------------	-------------------------------------

**Load Instructions: `lw $x, offset($y)`**

Instr. Fetch & PC Increm.	Read of Base Reg. \$y	ALU Op. ( $y + \text{offset}$ )	Read Mem. $M(y + \text{offset})$	Write Back of Destinat. Reg. \$x
------------------------------	--------------------------	------------------------------------	-------------------------------------	-------------------------------------

**Store Instructions: `sw $x, offset($y)`**

Instr. Fetch & PC Increm.	Read of Base Reg. \$y & Source \$x	ALU Op. ( $y + \text{offset}$ )	Write Mem. $M(y + \text{offset})$
------------------------------	---------------------------------------	------------------------------------	--------------------------------------

**Conditional Branch: `beq $x, $y, offset`**

Instr. Fetch & PC Increm.	Read of Source Regs. \$x and \$y	ALU Op. ( $x - y$ ) & ( $PC + 4 + \text{offset}$ )	Write PC
------------------------------	-------------------------------------	---	-------------

Figura 7: Cicli eseguiti da ogni operazione

### 1.1.3 Implementazione base di un MIPS

Vediamo ora come potrebbe essere una semplice implementazione di un *Data Path* in un MIPS. Come notiamo dalla Figura 9 abbiamo che la parte di memoria dedicata alle istruzioni (*Instruc-*

Instruction Type	Instruct. Mem.	Register Read	ALU Op.	Data Memory	Write Back	Total Latency
ALU Instr.	2	1	2	0	1	6 ns
Load	2	1	2	2	1	8 ns
Store	2	1	2	2	0	7 ns
Cond. Branch	2	1	2	0	0	5 ns
Jump	2	0	0	0	0	2 ns

Figura 8: Latenza delle diverse operazioni

*tion Memory*) è di sola lettura ed è separata dalla memoria dedicata ai dati (*Data Memory*). Inoltre abbiamo 32 registri organizzati in un *Register File* (RF) con 2 porte di lettura (le due frecce che escono sulla destra) e una porta in scrittura (la freccia in ingresso che punta al campo *Data*). La fase di *Instruction Fetch* invece richiede un adder il quale in uscita si connette al

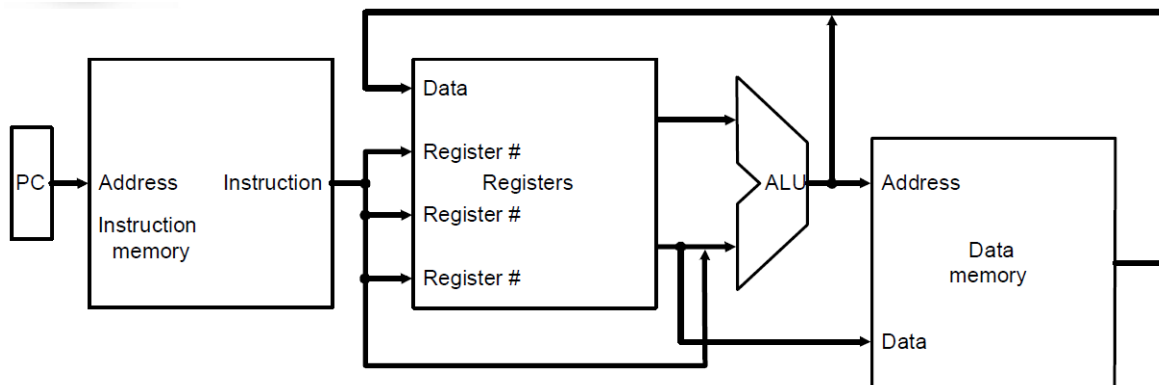


Figura 9: Esempio di implementazione di MIPS

PC mentre come ingressi riceve un valore costante 4 mentre all'altro ingresso riceve il valore corrente di del PC come possiamo vedere in Figura 10. Analizziamo ora in breve quale hardware

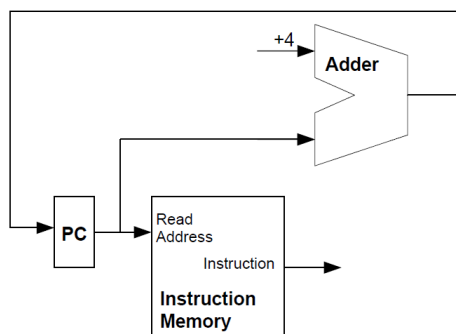


Figura 10: Hardware necessario per realizzare l'Instruction Fetch

è necessario per implementare le diverse operazioni che possono essere eseguite da un MIPS. Partiamo con l'analizzare un istruzione di tipo ALU come vediamo in Figura 11. Dal *Register File* escono due porte che sono connesse ad un'unità ALU la quale ha un uscita *Result* che si



connette alla porta di scrittura del *RF* e un'uscita *Zero* per indicare eventuali anomalie. Inoltre la ALU ha un ingresso *OP* che serve a selezionare il tipo di operazione. Per quanto riguarda

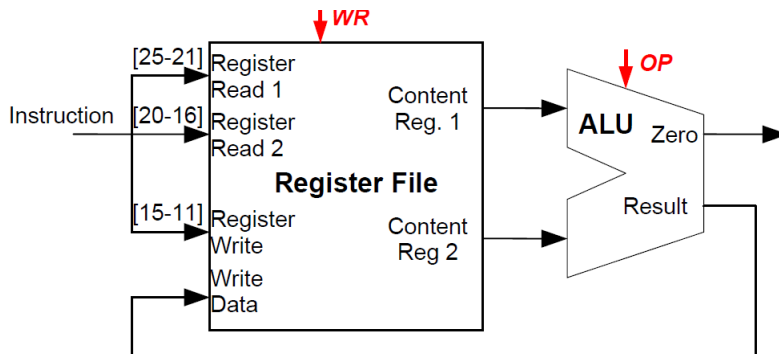


Figura 11: Hardware che implementa un'istruzione di tipo aritmetico

l'istruzione di *load* e quella di *store* sono molto simili come si può vedere da Figura 12 e da Figura 13. Nel caso della *load* la alu calcola l'indirizzo di memoria da leggere il quale viene inviato alla memoria e il risultato della lettura è registrato tramite la porta write data del *RF*. Nel caso della *store* invece la ALU calcola l'indirizzo di destinazione della scrittura e tramite la porta write del *Data Memory* viene copiato il valore del registro. In entrambi i casi un'unità

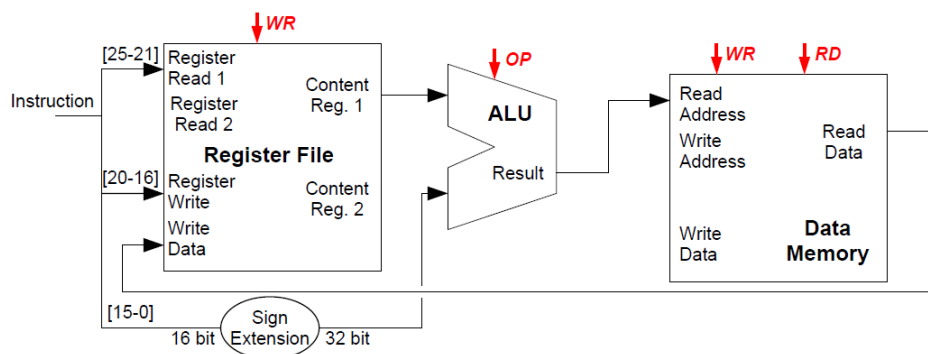


Figura 12: Hardware che implementa un'istruzione di tipo load

particolare si occupa di eseguire l'estensione del segno in caso di bisogno.

Stabiliamo ora come viene implementato il clock del circuito; possiamo avere due possibilità, la prima è avere un unico ciclo di clock lungo quanto il percorso critico necessario per eseguire l'istruzione di load (la più lunga), la seconda è avere un ciclo di clock lungo quanto un singolo passaggio in uno dei componenti prima analizzati.

Analizziamo innanzitutto il caso di singolo ciclo, in questo caso il ciclo dovrà avere una durata pari al tempo necessario per eseguire un'istruzione di load che come abbiamo visto è pari a  $T = 8ns$  ( $f = 125 MHz$ ). Assumiamo quindi che ogni istruzione verrà eseguita in un singolo ciclo di clock, ogni modulo verrà utilizzato una sola volta per clock e quei moduli che dovrebbero essere utilizzati più di una volta dovranno essere duplicati. Inoltre dobbiamo tener conto anche delle differenze tra i diversi tipi di istruzioni, infatti, all'ingresso di scrittura dell'*RF* possiamo avere dati provenienti da una ALU e quindi di lunghezza [15-11] bit oppure dati provenienti da una load/store con una lunghezza di [20-16] bit questo richiede un *Multiplexer* all'ingresso dei registri nel *RF*. In secondo luogo al secondo ingresso della ALU possiamo avere il dato proveniente da un registro nel caso di operazioni ALU oppure l'offset per le istruzioni di

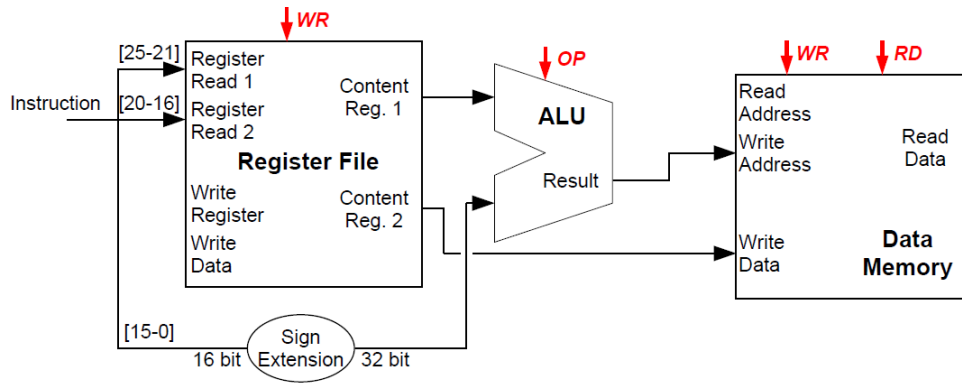


Figura 13: Hardware che implementa un'istruzione di tipo store

load/store, questo richiede un *MUX* al secondo ingresso della ALU. Infine i dati all'output del Destination Register possono arrivare sia dal risultato della ALU oppure dal Data Memory nel caso di load questo comporta l'utilizzo di un *MUX* all'ingresso in scrittura dei dati su RF. In Figura 14 vediamo l'implementazione completa di un MIPS a ciclo singolo con l'introduzione, oltre che dei *MUX* precedentemente specificati, anche di una ALU (parte alta della figura) che permette l'implementazione dei branch, e della logica di controllo (in rosso nella figura)

Veniamo ora al caso in cui il ciclo di clock sia di lunghezza pari al tempo necessario per un singolo modulo  $T = 2ns$  questo significa che per eseguire un'istruzione di load sono necessari 5 cicli di clock per un totale di  $10ns$ . Ogni fase dell'istruzione richiede un ciclo di clock ma questo permette la condivisione dei moduli tra diverse istruzioni in differenti cicli di clock, anche se questo richiede l'inserimento di registri tra un'unità e l'altra.

## 1.2 Pipelining

Il pipelining è una tecnica di ottimizzazione basata sull'esecuzione multipla sovrapposta di istruzioni sequenziali. L'idea fondamentale è quella di sfruttare il parallelismo intrinseco delle istruzioni sequenziali in quanto l'esecuzione di una istruzione è suddivisa in fasi differenti (*pipelines stages*) che richiedono soltanto una piccola frazione di tempo per essere completate. I diversi stati sono connessi in sequenza nella pipeline, un'istruzione entra da una parte procede attraverso i diversi stadi e esce all'altro capo come in una catena di montaggio.

I vantaggi di questa tecnica è che è completamente trasparente al programmatore, inoltre come in una catena di montaggio, il tempo necessario per eseguire un'istruzione è uguale al caso in cui l'istruzione sia eseguita senza pipeline; quello che la pipeline fa è incrementare il numero di istruzioni eseguite contemporaneamente e perciò aumentare la frequenza di completamento come vediamo in Figura 15

Il tempo necessario per far avanzare un'istruzione di una fase corrisponde ad un ciclo di clock, le diverse fasi perciò devono essere *sincronizzate*, il periodo di clock deve essere uguale al tempo di esecuzione della fase più lenta (nel nostro esempio  $2ns$ ). L'obiettivo è quello di bilanciare la lunghezza di ogni fase della pipeline in modo da avere uno *speedup ideale* uguale al numero di fasi della pipeline.

Nel caso ideale vediamo come la pipeline sia più efficiente sia dell'architettura a singolo ciclo che a quella multi-ciclo viste in precedenza. Nel caso di una CPU1 non pipeline con un unico ciclo di clock della durata di  $8ns$  contro una CPU2 con una pipeline a 5 stadi e ciclo di  $2ns$  abbiamo che:

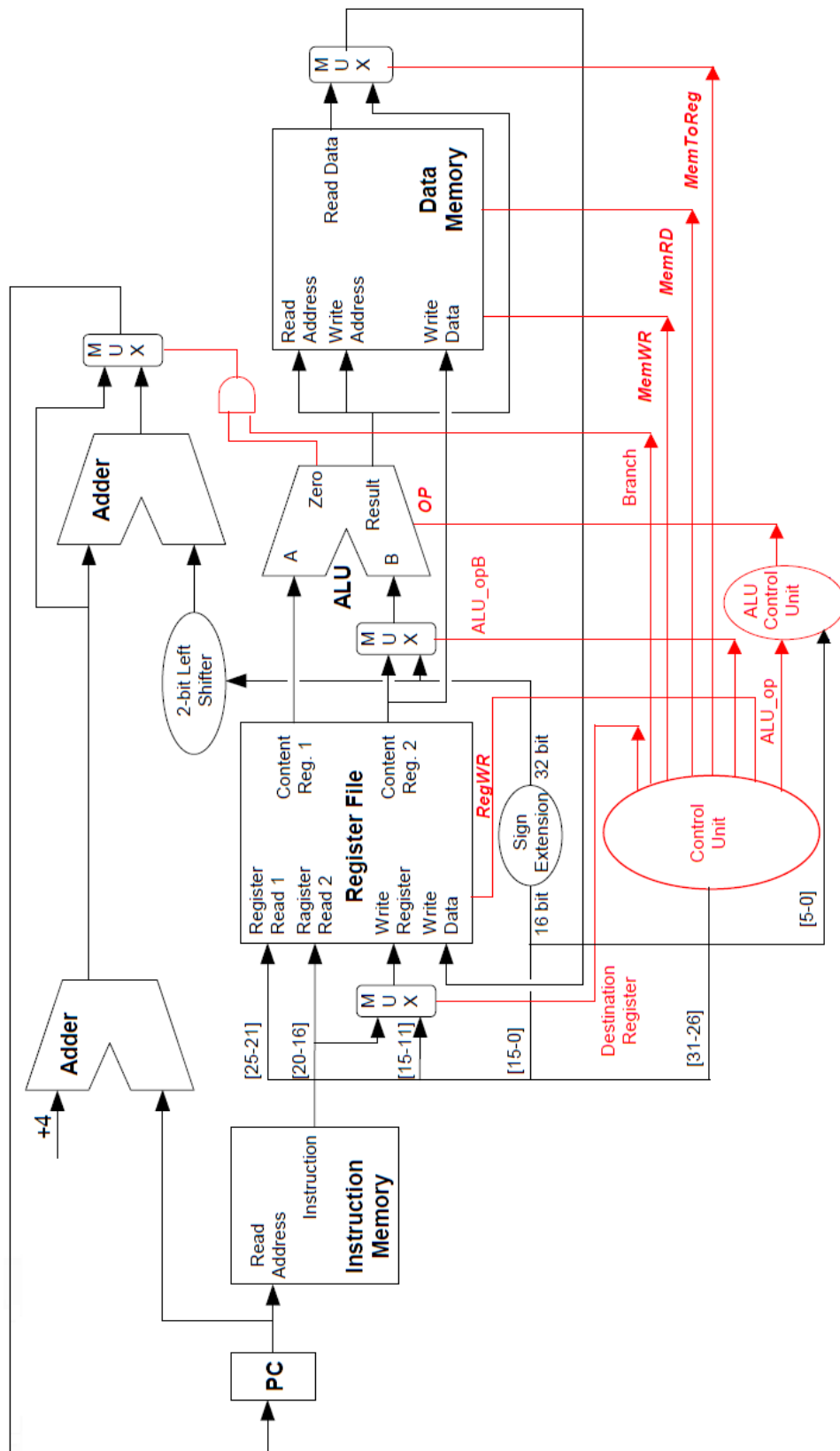


Figura 14: MIPS a singolo ciclo con logica di controllo

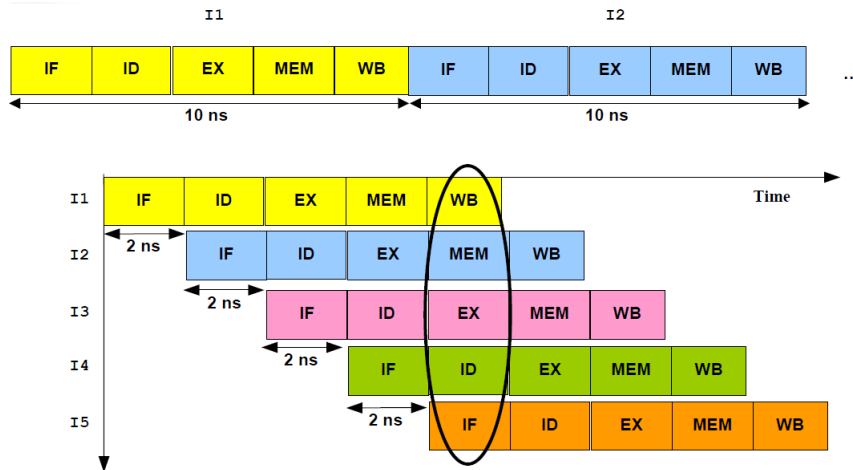


Figura 15: Confronto tra esecuzione sequenziale e pipelined

- la *latenza* ovvero il tempo necessario per eseguire una istruzione è peggiore nel caso di CPU2: 8ns vs 10ns
- il *throughput* tuttavia è notevolmente migliorato: 1 istruzione/8ns vs 1 istruzione/2ns

Nel caso di CPU3 multi-ciclo senza pipeline contro un'architettura CPU2 come quella descritta in precedenza abbiamo:

- la *latenza* resta invariata: 10 ns
- il *throughput* cresce di ben 5 volte: 1 istruzione/10ns vs 1 istruzione/2ns

### 1.2.1 Implementazione di una Pipeline

Innanzitutto vediamo quali fasi devono attraversare ciascuna operazione in quanto non tutte le fasi sono necessarie per tutte le operazioni, uno schema riassuntivo è specificato in Figura 16.

La divisione dell'esecuzione di una istruzione in 5 fasi implica che in ogni ciclo di clock cinque

IF Instruction Fetch	ID Instruction Decode	EX Execution	ME Memory Access	WB Write Back
ALU Instructions: <b>op \$x,\$y,\$z</b>				
Instr. Fetch & PC Increm.	Read of Source Regs. \$y and \$z	ALU Op. (\$y op \$z)		Write Back Destin. Reg. \$x
Load Instructions: <b>lw \$x,offset(\$y)</b>				
Instr. Fetch & PC Increm.	Read of Base Reg. \$y	ALU Op. (\$y+offset)	Read Mem. M(\$y+offset)	Write Back Destin. Reg. \$x
Store Instructions: <b>sw \$x,offset(\$y)</b>				
Instr. Fetch & PC Increm.	Read of Base Reg. \$y & Source \$x	ALU Op. (\$y+offset)	Write Mem. M(\$y+offset)	
Conditional Branches: <b>beq \$x,\$y,offset</b>				
Instr. Fetch & PC Increm.	Read of Source Regs. \$x and \$y	ALU Op. (\$x-\$y) & (PC+4+offset)	Write PC	

Figura 16: Fasi della pipeline necessarie ad ogni istruzione

istruzione sono in esecuzione questo comporta la necessità di inserire dei *registri* tra una fase e l'altra della pipeline per separare i diversi stage.

In Figura 17 vediamo una possibile implementazione di un'architettura MIPS pipelined con l'introduzione dei registri tra le fasi (in verde).

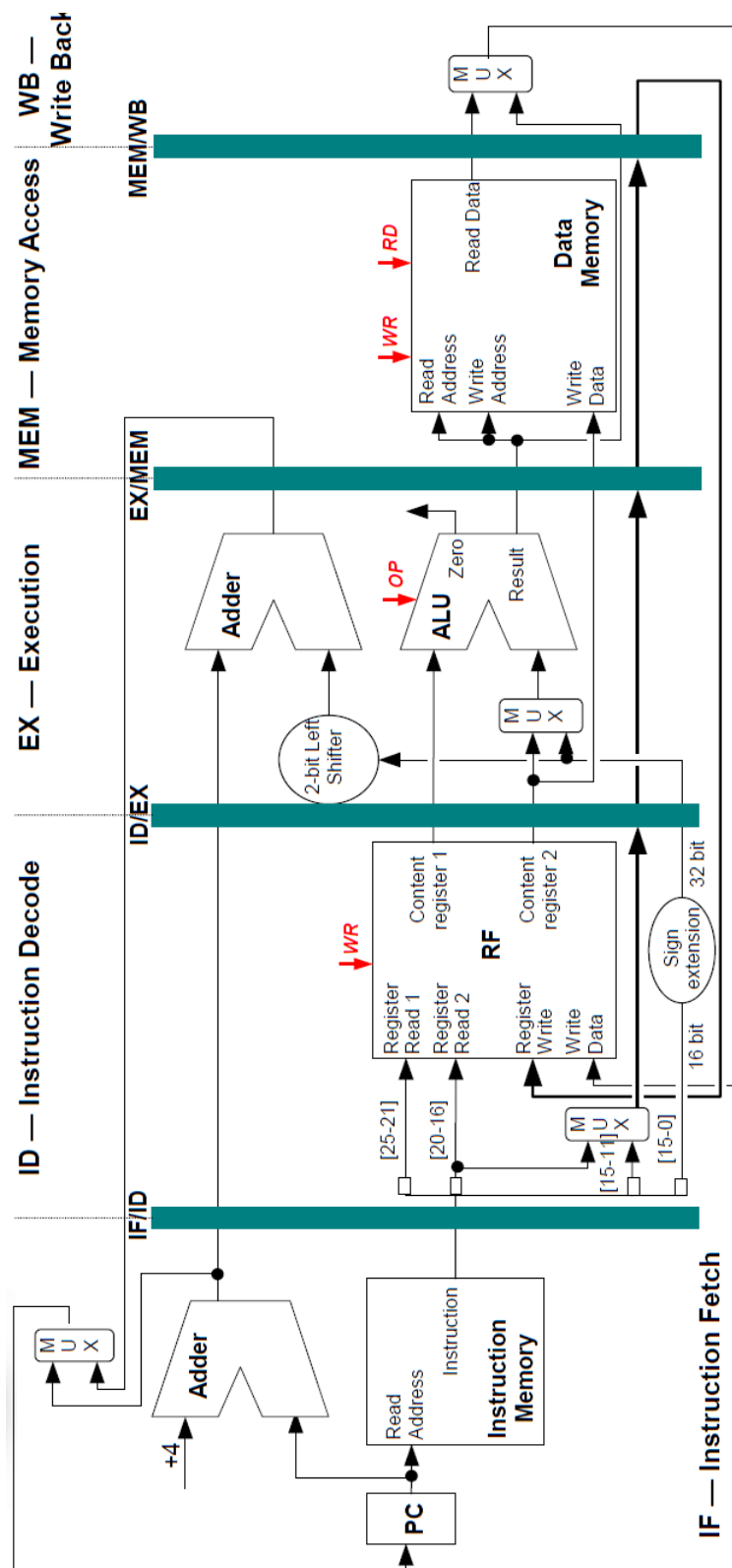


Figura 17: Schema di un MIPS con pipeline

### 1.3 Il problema del "Hazard"

Si ha un *hazard* quando vi è una dipendenza tra istruzioni diverse e la sovrapposizione dovuta al pipeline cambia l'ordine di dipendenza sugli operandi. Hazard previene l'esecuzione della prossima istruzione nel ciclo di clock designato ma così facendo riduce le performance allontanandole dallo speedup ideale.

Possiamo distinguere tre classi di *hazard*:

- **Structural Hazards:** si ha quando diverse istruzioni cercano di utilizzare la stessa risorsa simultaneamente (stessa memoria per istruzioni e dati)
- **Data Hazards:** si ha quando si cerca di utilizzare un risultato prima che questo sia pronto (istruzione dipendente dalla precedente che è nella pipeline)
- **Control Hazards:** si ha quando si deve prendere una decisione sulla esecuzione della prossima istruzione prima della valutazione di una condizione (branch condizionali)

Tra questi tre tipi di hazard il primo non può presentarsi nelle architetture MIPS in quanto lo spazio di memoria dedicato alle istruzioni e quello dedicato ai dati sono fisicamente separati.

#### 1.3.1 Data Hazard

Per quanto riguarda il data hazard si verifica quando sono in esecuzione nella pipeline due o più istruzioni *dipendenti*.

```
sub    $2, $1, $3
and    $12, $2, $5    #1° operando dipende dalla sub
or     $13, $6, $2    #2° operando dipende dalla sub
add    $14, $2, $2    #1° & 2° operando dipendono dalla sub
sw     $15, 100($2)   #Il registro base dipende dalla sub
```

Come vediamo dall'esempio qui sopra e dalla sua esecuzione in Figura 18 abbiamo che le istruzioni successive alla *sub* debbono aspettare che la prima istruzione arrivi nella fase di *write-back* prima di poter utilizzare il dato come avviene per l'ultima istruzione evidenziata da una freccia verde. Esistono diversi meccanismi per far sì che queste dipendenze vengano soddisfatte, le

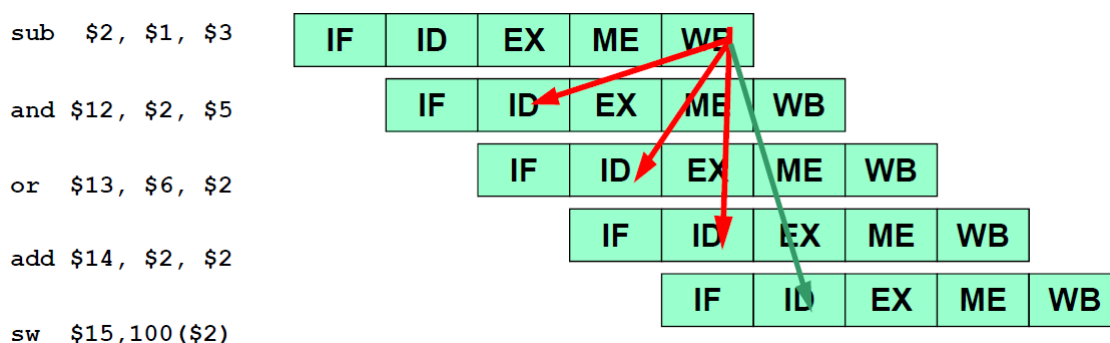


Figura 18: Esempio di data hazard

principali si possono suddividere in due categorie:

- **Tecniche di compilazione:** in questa categoria rientra il re-scheduling delle operazioni che consiste nell'inserire istruzioni indipendenti tra le istruzioni correlate in modo da

permettere il calcolo dei valori necessari; nel caso non sia possibile inserire altre operazioni il compilatore inserisce delle **nop** ovvero delle operazioni che non fanno nulla *no operation*

- **Tecniche hardware:** in questa categoria rientrano la possibilità di inserire delle *bubbles* o degli stalli oppure le tecniche di *Data Forwarding* e di *Bypassing*

Vediamo innanzi tutto un esempio di inserimento di **nop** in Figura 19 Come vediamo l'inserimento

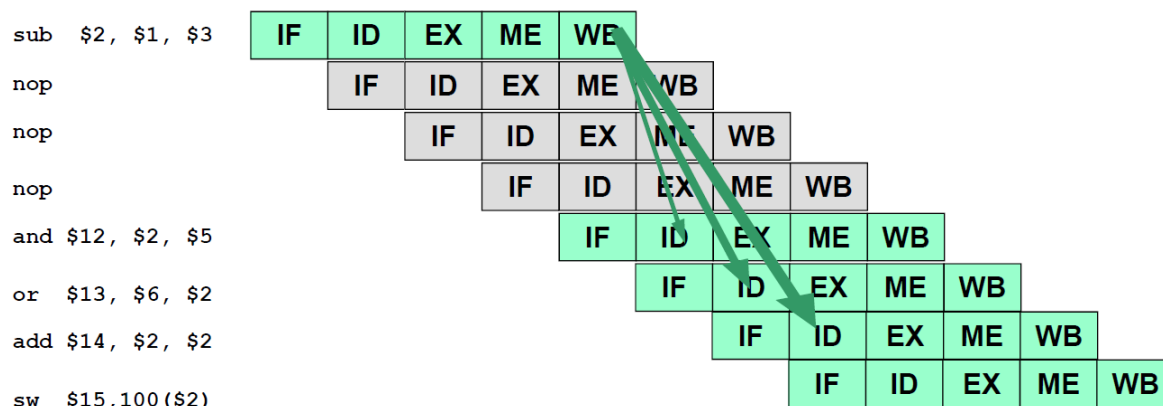


Figura 19: Esempio di uso **nop**

di **nop** peggiora lo speedup ideale, cosa che invece non succede se si applicano le tecniche di re-scheduling in quanto non vengono inserite istruzioni inutili nell'esecuzione delle istruzioni ma viene modificato semplicemente l'ordine nel quale vengono eseguite.

Il caso di inserimento di stalli è molto simile a quello di inserimento delle **nop** la differenza sta nel fatto che si ferma l'esecuzione dell'istruzione dipendente il tempo necessario affinché l'istruzione in esecuzione renda disponibile il dato come vediamo in Figura 20, anche in questo caso abbiamo un peggioramento dello speedup ideale.

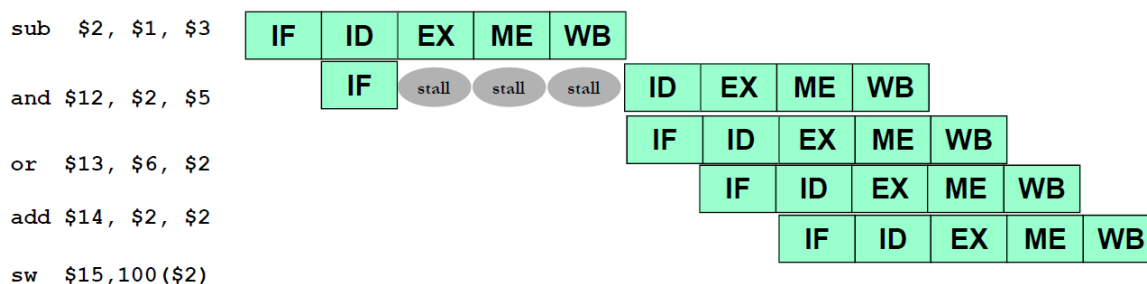


Figura 20: Esempio di uso degli stalli

**Data Forwarding** Il *data forwarding* è una tecnica hardware che comporta l'utilizzo dei risultati temporanei immagazzinati nei registri della pipeline, per fare ciò abbiamo bisogno di aggiungere dei *multiplexer* all'ingresso della ALU per selezionare la provenienza dei dati. In Figura 21 vediamo quali sono i collegamenti necessari per l'esecuzione di istruzioni aritmetiche dipendenti; questi path sono tre:

- **EX/EX path:** in figura da ALU ad ALU che risolve il problema di due istruzioni consecutive nella quale la seconda necessita del risultato della prima. Nell'esempio precedente la dipendenza **sub**→**and**



- **MEM/EX path:** in questo caso si risolve la dipendenza tra la prima e la terza istruzione (sub→or)
- **MEM/ID path:** risolve la dipendenza tra la prima e la quarta istruzione (sub→add)

Con l'introduzione di questi path si riesce a risolvere tutti i problemi di dipendenza per quanto riguarda le istruzioni di tipo aritmetico. In Figura 22 vediamo una possibile implementazione hardware di una pipeline con sistema di forwarding path. Esiste ancora una situazione però

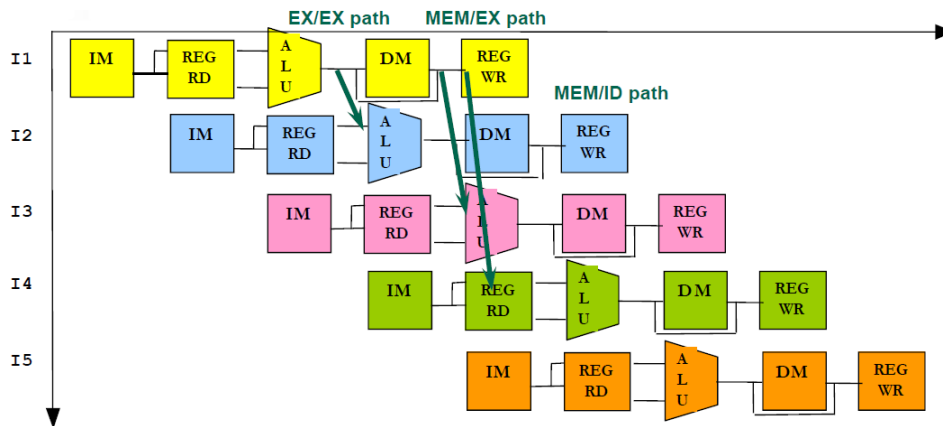


Figura 21: Esempio di forwarding path

in cui è necessario inserire degli stalli, questa situazione è dovuta alla sequenza di istruzioni seguenti:

```
L1: lw  $s0, 4($t1)    # $s0 <- M[4 + $t1]
L2: add $s5, $s0, $s1  # 1° operand depends from L1
```

Questa dipendenza si crea in quanto il dato viene scritto in **\$s0** solo nella fase di WB mentre viene letto durante la fase di ID. In questo caso non si può fare molto se non sfruttare il forwarding path MEM/EX già analizzato prima anche se comunque è necessario introdurre uno stallo per risolvere la dipendenza. Nel caso invece in cui la dipendenza sia tra un'istruzione di *load* e una di *store* si può risolvere la dipendenza aggiungendo un *forwarding path* tra le due fasi di MEM. Questo path permette di risolvere la dipendenza senza dover introdurre altri stalli come si può vedere in Figura 23. Con l'architettura attuale, come abbiamo visto, nel caso di dipendenza tra una *load* ed un'istruzione aritmetica che legge un registro è necessario introdurre uno stallo; in quanto l'accesso in lettura avviene durante la fase di ID mentre la scrittura avviene durante la fase di WB. Nel caso, invece, di *pipeline ottimizzata* possiamo assumere che la fase di lettura avviene nella seconda metà del ciclo di clock mentre la fase di scrittura nella prima metà; in questo modo nel caso in cui lettura e scrittura facciano riferimento allo stesso registro nello stesso ciclo di clock non è più necessario inserire degli stalli, e si può inoltre eliminare il forwarding path tra MEM e ID.

### 1.3.2 Altri tipi di Data Hazard

Fino ad ora abbiamo analizzato solo un tipo di dipendenza sui dati, questo tipo di dipendenza è chiamato **RAW (Read After Write)**, e si ha quando l'istruzione  $n+1$  cerca di leggere un registro prima che l'istruzione  $n$  abbia finito di scrivere tale registro.

Esistono tuttavia altri due tipi di *data hazard* che sono:

- Write After Write (WAW)
- Write After Read (WAR)

La dipendenza di tipo WAW si ha quando un'istruzione  $n+1$  tenta di scrivere un registro il quale non è ancora stato scritto dall'istruzione  $n$ . Tale tipo di dipendenza avviene solo nel caso in cui la nostra pipeline preveda la possibilità di fasi di memorizzazione o di esecuzione multi-ciclo come mostrato in Figura 24 e 25 le quali portano alla terminazione delle istruzioni fuori ordine. Per quanto riguarda le dipendenze di tipo WAR si hanno quando l'istruzione  $n+1$  tenta di scrivere un registro prima che questo sia stato letto da un'istruzione  $n$ , nel caso di architettura MIPS però tale tipo di dipendenza non può mai verificarsi in quanto la lettura avviene nella fase ID mentre la scrittura nella fase WB.

## 1.4 Analisi delle performance

L'utilizzo della pipeline aumenta il throughput della CPU ma non riduce il tempo di esecuzione della singola istruzione, anzi solitamente aumenta la latenza di ogni istruzione bisogna quindi bilanciare il numero di fasi con l'overhead dovuto alla pipeline.

Definito  $IC = Instruction\ Count$  ovvero il numero di istruzioni eseguite, possiamo determinare il numero di cicli di clock necessari per completare queste operazioni. Tale valore è uguale a:

$$\#Clock\ Cycle = IC + \#Stall\ Cycles + 4$$

Dividendo tale valore per il numero di operazioni otteniamo:

$$\begin{aligned} CPI &= Clock\ Per\ Instruction = \#Clock\ Cycle / IC = \\ &= (IC + \#Stall\ Cycles + 4) / IC \end{aligned}$$

$$MIPS = f_{clock} / (CPI * 10^6)$$

Come visto fino ad ora la CPI ideale per la pipeline è 1 ma gli stalli degradano le performance. Abbiamo così che la CPI media è data da:

$$\begin{aligned} Ave.\ CPI &= Ideal\ CPI + \#Stall\ per\ Instruction \\ &= 1 + \#Stall\ per\ Instruction \end{aligned}$$

Possiamo misurare il miglioramento delle performance dato dall'introduzione della pipeline come

$$\begin{aligned} Pipeline\ SpeedUp &= \frac{Ave.\ Exec.\ Time\ Unpipelined}{Ave.\ Exec.\ Time\ Pipelined} = \\ &= \frac{Ave.\ CPI\ Unp.}{Ave.\ CPI\ Pipe} \times \frac{Clock\ Cycle\ Unp.}{Clock\ Cycle\ Pipe} \end{aligned}$$

Se ignoriamo l'overhead sul tempo di clock e assumiamo che i diversi stage siano perfettamente bilanciati possiamo ridefinire lo speedup come

$$SpeedUp_{pipeline} = \frac{Ave.\ CPI\ Unp.}{1 + \#Stall\ per\ Instruction}$$

Nel caso ideale nel quale tutte le istruzioni richiedano lo stesso numero di cicli questi sono uguali al numero di fasi della pipeline e possiamo riscrivere la precedente come:

$$SpeedUp_{pipeline} = \frac{Pipeline\ Depth}{1 + \#Stall\ per\ Instruction}$$

Nel caso ideale in cui non ci siano stalli vediamo come le performance migliori tanto è più profonda (maggiore numero di fasi) la pipeline.

Nel caso in cui si abbiano dei salti condizionati le performance peggiorano in base alla penalità del branch, infatti:

$$SpeedUp_{pipeline} = \frac{Pipeline\ Depth}{1 + Branch\ Frequency \times Branch\ Penalty}$$

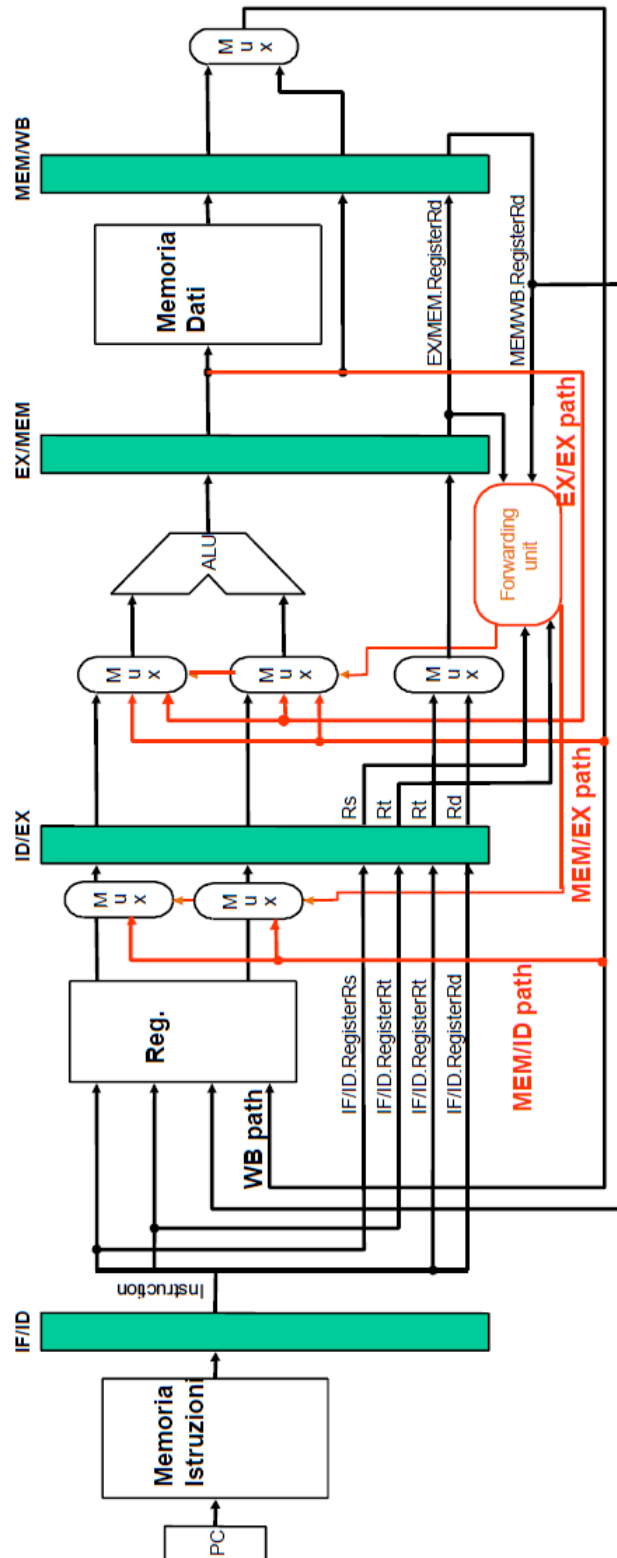


Figura 22: Schema MIPS con forwarding

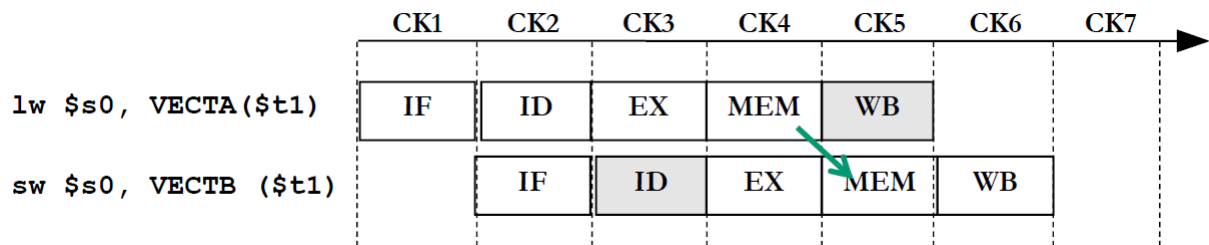


Figura 23: Forwarding path tra due fasi di memorizzazione successive.

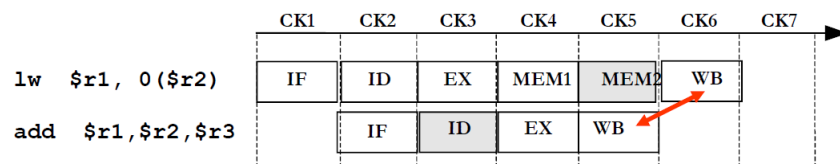


Figura 24: Fase di memorizzazione multiciclo che porta alla creazione di dipendenze di tipo WAW

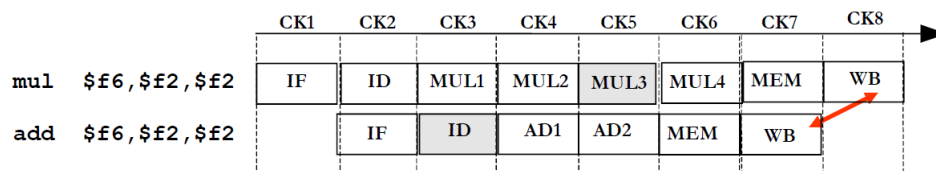


Figura 25: Fase di esecuzione multiciclo che porta alla creazione di dipendenze di tipo WAW

op	rs	rt	address
6 bit	5 bit	5 bit	16 bit

Figura 26: Esempio di istruzione di branch

IF Instruction Fetch	ID Instruction Decode	EX Execution	ME Memory Access	WB Write Back
-------------------------	--------------------------	-----------------	---------------------	------------------

**beq \$x,\$y,offset**

Instr. Fetch & PC Increm.	Register Read \$x e \$y	ALU Op. (\$x-\$y) & (PC+4+offset)	Write of PC
------------------------------	----------------------------	--------------------------------------	----------------

Figura 27: Suddivisione dell'esecuzione di un'istruzione di salto nelle varie fasi di una pipe

## 2 Tecniche di predizione dei salti

Come abbiamo visto nel Capitolo 1 i branch condizionati sono istruzioni di salto che vengono eseguite soltanto se viene soddisfatta la condizione. L'indirizzo di destinazione del branch viene sostituito nel program counter al posto dell'indirizzo dell'istruzione sequenziale successiva. Nel caso di ambiente MIPS possiamo distinguere due tipi di branch:

- **beq**: *branch on equal* che richiede che i valori nei registri da confrontare siano uguali.
- **bne**: *branch on not equal* che richiede che i valori nei registri da confrontare siano diversi.

Come abbiamo visto nel capitolo precedente le istruzioni di salto condizionato sono nel formato *I-Format* e un'istruzione di questo tipo è suddivisa nei diversi campi come mostrato in Figura 26 Dove il campo **address** indica l'indirizzo relativo rispetto al program counter che punta all'etichetta di salto.

Questo tipo di istruzione sfrutta solo quattro dei cinque stadi della pipeline come mostrato in Figura 27; durante l'instruction fetch si recupera l'istruzione da eseguire e si aggiorna il program counter all'istruzione sequenziale successiva, successivamente durante la fase di instruction decode si leggono i due registri da comparare. Durante la fase di *execution* la ALU compara i due registri e calcola il valore di destinazione del salto. Durante la fase *Memory Access* si decide in base al valore della comparazione effettuata dalla ALU se aggiornare il PC con il valore del salto.

### 2.1 Il problema del Control Hazard

Il *control hazard* è il problema di decidere quale istruzione eseguire prima che la condizione di salto sia valutata. I problemi di *control hazard* nascono ogni qualvolta nella pipeline sia necessario modificare il valore del PC. Tali problemi riducono perciò la velocità della pipeline riducendo lo speedup ideale a causa di introduzioni di stalli nella pipeline.

Per alimentare la pipeline è necessario prelevare una nuova istruzione ad ogni ciclo di clock ma la decisione se effettuare o non effettuare un salto avviene solo durante lo stage *MEM*. Questo ritardo nel determinare l'istruzione successiva corretta è chiamato *Control Hazard* o *Conditional*

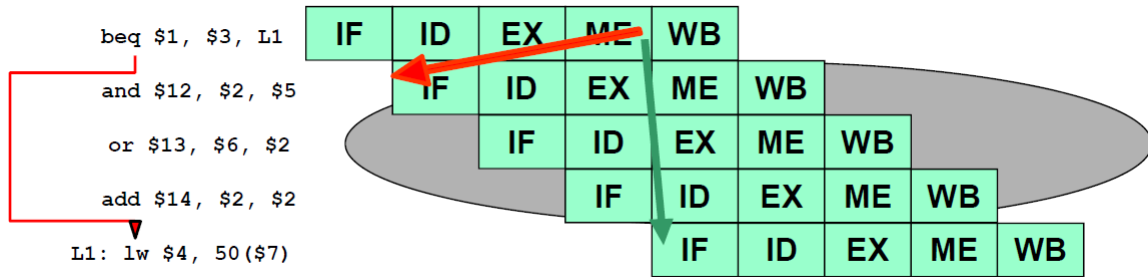


Figura 28: Esempio di esecuzione di un'istruzione di salto

### Branch Hazard.

Analizziamo ora l'esempio di Figura 28 in questo esempio la prima istruzione è un salto condizionato che viene valutato solo nella fase si MEM; durante l'esecuzione di tale istruzione vengono prelevate anche le tre istruzioni successive per continuare ad alimentare la pipeline. Se il salto non viene eseguito l'esecuzione è corretta e può proseguire, nel caso in cui, invece, il salto venga eseguito allora diventa necessario effettuare il *flush* delle tre istruzioni prelevate durante l'esecuzione del branch. Una possibile soluzione al problema è quella di attendere la decisione del salto prima di effettuare qualsiasi altra operazione; questo comporta l'inserimento di stalli nella pipeline; più precisamente sono necessari:

- tre stalli senza forwarding
- due stalli con forwarding

Nel caso in cui il salto non venga eseguito, tuttavia, la penalità di tre cicli di stallo non è giustificata. Un'altra soluzione è quella di assumere che il salto non sia mai eseguito e quindi scartare le tre istruzioni nel caso in cui il salto venga preso.

Una terza soluzione è quella di aggiungere delle risorse hardware per permettere di:

- comparare i registri
- calcolare l'indirizzo di destinazione del branch
- aggiornare il valore del PC

il prima possibile nella catena della pipeline. Nei processori MIPS tutto questo avviene durante lo stage ID come mostrato in Figura 29 Utilizzando queste tecniche si riesce a ridurre al minimo il costo per recuperare la corretta esecuzione di un branch nel caso di scelta sbagliata, riducendo a uno il numero degli stalli da introdurre.

Tuttavia queste tecniche comportano tuttavia una riduzione delle performance quantificabile tra il 10% e il 30% in base alla frequenza dei salti. Tali perdite di performance possono essere ridotte ulteriormente tramite alcune tecniche.

## 2.2 Tecniche di predizione dei salti

In generale il problema dei salti diventa importante quando si tratta di processori con delle pipeline profonde, dove il costo di una predizione errata è molto alto. Lo scopo principale delle tecniche di predizione dei salti è quello di predire il prima possibile il risultato di un'istruzione di salto.

Le performance di una tecnica di predizione si possono misurare in:

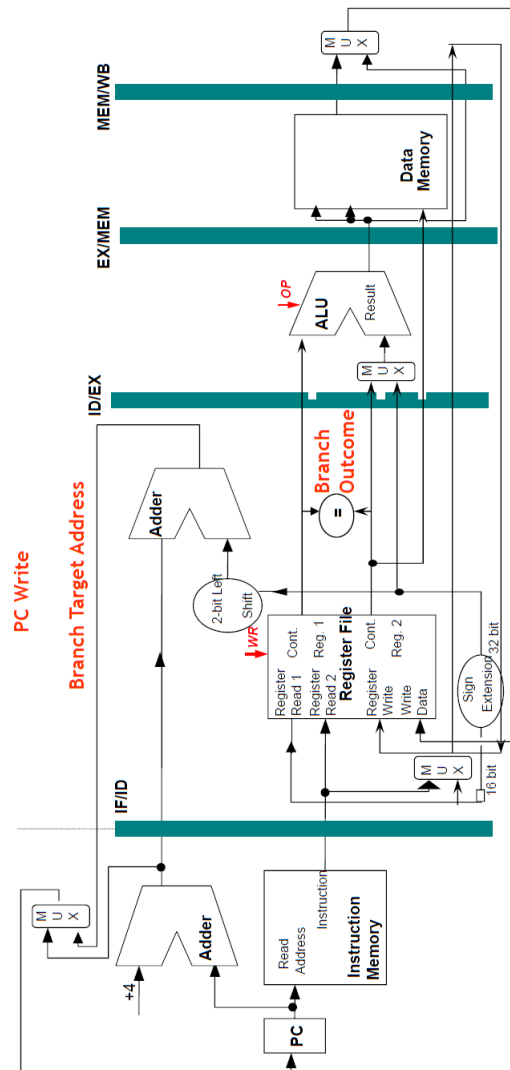


Figura 29: Hardware aggiuntivo per risolvere i problemi di controllo



- **Accuratezza:** misurata in termini di percentuale di predizioni sbagliate.
- **Costo:** misurato come tempo perso nel caso di predizione sbagliata.

Le tecniche di predizione dei salti possono essere suddivise in due categorie:

- *Tecniche di predizione statiche:* le azioni intraprese per il branch sono prefissate e uguali per tutta l'esecuzione e determinate a tempo di compilazione.
- *Tecniche di predizione dinamiche:* in questo caso le decisioni variano a seconda dell'esecuzione.

### 2.2.1 Tecniche di predizione statiche

Le tecniche di predizione statiche sono utilizzate soprattutto in quei processi dove ci si aspetta che i salti siano altamente predicibili. Alcune tecniche statiche di predizione dei salti sono:

- Branch Always Not Taken (Predicted-Not-Taken)
- Branch Always Taken (Predicted-Taken)
- Backward Taken Forward Not Taken (BTFNT)
- Profile-Driven Prediction
- Delayed Branch

**Branch Always Not Taken** In questa particolare tecnica assumiamo che il salto non venga mai intrapreso e le istruzioni vengono prelevate sequenzialmente e il flusso prosegue come se il salto non venga intrapreso. Se la condizione nello stage ID non viene soddisfatta la predizione è corretta e perciò non abbiamo perdita di performance.

Se la condizione nello stage ID risulta soddisfatta allora la predizione è errata e il salto viene effettuato. A questo punto dobbiamo effettuare il flush delle successive istruzioni che sono già state messe in esecuzione sostituendole con delle `nop` e riprendere l'esecuzione inserendo nella pipeline la prima istruzione del salto. Tutto questo porta ad una penalità di un ciclo di clock.

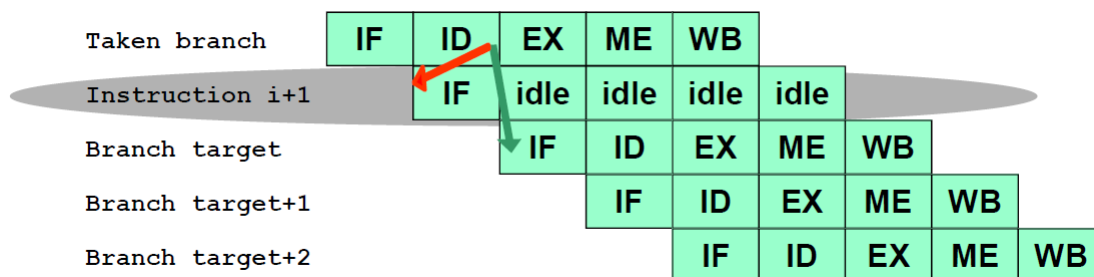


Figura 30: Esempio di penalità dovuto ad una predizione sbagliata

**Branch Always Taken** In alternativa al tipo di predizione precedente si può considerare che ogni salto sia sempre eseguito. Ogni qualvolta che un salto è decodificato e il suo indirizzo di destinazione è calcolato allora si assume che il salto sia eseguito e si introducono nella pipeline le istruzioni puntate dall'indirizzo di destinazione. Questo tipo di previsione ha senso in quelle pipeline dove l'indirizzo target è calcolato prima della comparazione dei registri.

Nelle pipeline di tipo MIPS noi non conosciamo l'indirizzo di destinazione prima della valutazione delle condizioni di salto così non vi è alcun vantaggio dall'utilizzo di questa tecnica.

**Backward Taken Forward Not Taken (BTFNT)** La predizione di questa tecnica si basa sulla direzione del salto, ovvero se i salti sono all'indietro sono previsti come eseguiti (come ad esempio nei cicli) salti in avanti sono considerati come non eseguiti.

**Profile-driven Prediction** Questo tipo di predizione si basa su dati raccolti da precedenti esecuzioni del programma utilizzando alcune funzioni del compilatore.

**Delayed Branch Technique** In questo tipo di tecnica il compilatore schedula una particolare istruzione indipendente dal salto in un campo chiamato **branch delay slot**. L'istruzione in questo slot viene eseguita ogni qualvolta che il salto viene eseguito oppure no. Se assumiamo il ritardo dovuto ad un salto pari ad un ciclo di clock allora gli slot necessari per le istruzioni sono uno. Un esempio di questa tecnica è mostrato in Figura 31 dove nel *branch delay slot* viene inserita un'istruzione di **add** indipendente dal ciclo. Sia nel caso che il salto sia eseguito sia nel

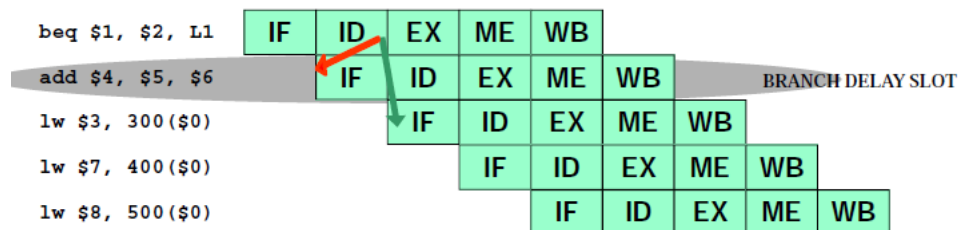


Figura 31: Esempio di utilizzo del branch delay slot

caso non sia eseguito l'istruzione dopo quella di salto è sempre quella del *delay slot* tuttavia nel caso il salto sia eseguito dopo l'istruzione del *delay slot* l'esecuzione prosegue con le istruzioni del salto viceversa nel caso non sia eseguito allora l'esecuzione prosegue con le istruzioni successive. Il compilatore deve essere in grado di selezionare l'istruzione da inserire nel delay slot in modo che essa sia valida ed utile. Ci sono quattro modi per selezionare tale istruzione:

- From Before
- From Target
- From Fall-Through
- From After

La tecnica *From Before* prevede di selezionare un'istruzione indipendente selezionata tra quelle che precedono il salto in tale modo l'istruzione viene sempre eseguita. Il metodo *From Target* prevede la copia dell'istruzione puntata dal salto; questa tecnica è preferibile quando il salto ha un'alta probabilità di essere eseguito. Un esempio di utilizzo di questa tecnica è mostrato

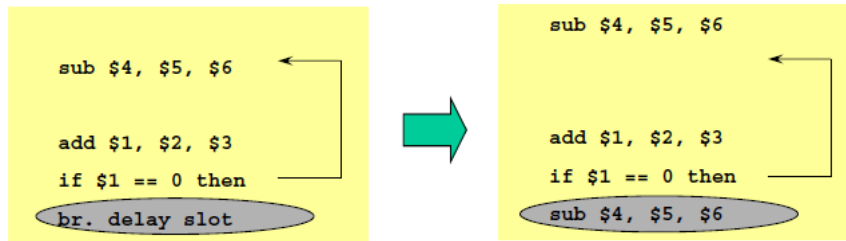


Figura 32: Esempio di selezione dell'istruzione *From Target*

in Figura 32. La tecnica *From Fall-Through* è contrapposta alla tecnica *From Target* infatti questa prevede che l'istruzione selezionata per il delay slot sia la prima istruzione che verrebbe prelevata nel caso di salto non eseguito come si vede dalla Figura 33. Questo tipo di tecnica è

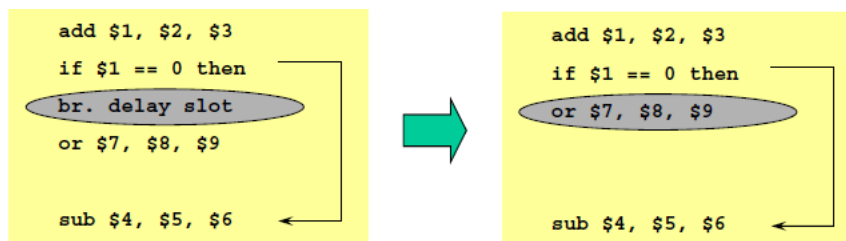


Figura 33: Esempio di uso della tecnica *From Fall-Through*

preferibile quando il salto ha un alta probabilità di essere non eseguito. Perché le ultime due tecniche risultino corrette è necessario che le istruzioni eseguite quando il salto non prende la direzione prevista risultino ininfluenti rispetto alla normale esecuzione del programma. Questo è possibile ad esempio se l'istruzione nel delay slot agisce su dei registri che sono inutilizzati nel caso di risultato inaspettato del salto.

In generale il compilatore è in grado di assegnare il 50% dei *delay slot* con istruzioni valide e utili, all'altro 50% viene riempito con istruzioni di tipo `nop`. Nel caso di pipeline più profonda il tempo necessario per valutare il salto è maggiore di un ciclo di clock e di conseguenza aumenta il numero di delay slot da riempire; diventa sempre più difficile perciò riempire tali slot con istruzioni valide e utili. Le principali limitazioni che nascono sono dovute alle restrizioni sulle istruzioni che possono essere schedate e sull'abilità del compilatore di predire staticamente il risultato del salto.

Per migliorare l'abilità del compilatore di riempire i *delay slot* molti processori hanno introdotto il **canceling or nullifying branch** ovvero salti nei quali è anche compresa la predizione della direzione del salto. Quando la predizione è verificata allora il contenuto del *delay slot* è eseguito in caso contrario viene eseguita una `nop`.

## 2.2.2 Tecniche di predizione dinamiche

L'idea di base in questo tipo di tecnica è quella di utilizzare il risultato di esecuzioni di salti passate per predire i salti futuri. Possiamo utilizzare dell'hardware per predire dinamicamente il risultato del salto; la predizione dipende dal comportamento dei salti durante l'esecuzione. Il meccanismo di predizione dinamica si basa su due meccanismi che interagiscono tra loro:

- **Branch Outcome Predictor:** che tenta di predire la direzione del branch.

- **Branch Target Predictor:** che calcola l'indirizzo di destinazione del salto nel caso in cui la condizione del salto abbia un risultato positivo.

Questi due moduli sono usati dall'unità di Instruction Fetch per predire la prossima istruzione da prelevare dall'I-Cache. Nel caso in cui il salto non venga eseguito il PC viene semplicemente incrementato, nel caso in cui, invece il branch venga preso il branch target predictor fornisce l'indirizzo di destinazione. Esiste inoltre una tabella chiamata **Branch History Table** la quale contiene un bit per ogni predizione passata che indica se il salto è stato preso oppure no. L'indice della tabella è basato su di una piccola porzione dell'indirizzo dell'istruzione di salto. Se la previsione è corretta e si prosegue nella direzione della previsione. Se la previsione risulta sbagliata il bit di previsione viene invertito e riportato nella tabella. Ogni accesso in tabella è un *hit* anche se tuttavia il bit di predizione può essere stato modificato da un altro salto con la stessa porzione di indirizzo. Una predizione errata avviene quando una predizione risulta

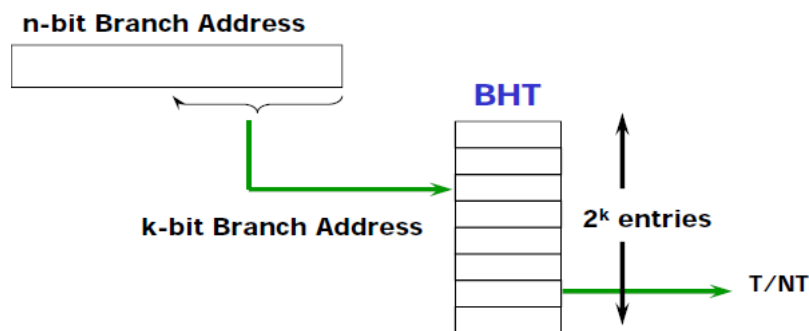


Figura 34: Esempio di *Branch History Table*

sbagliata oppure quando un indice è puntato da due differenti salti e la previsione si riferisce all'altro salto, per risolvere questo problema è sufficiente incrementare il numero di righe della BHT o utilizzare una funzione di hash.

Nel caso di una BHT ad un solo bit e considerando per esempio un salto di un loop che solitamente è eseguito la BHT sbaglierà predizione due volte, nel caso non sia eseguito e nel caso in cui venga eseguito il salto subito dopo non essere stato eseguito. Per meglio specificare i due casi abbiamo:

- All'ultima iterazione quando il salto non viene eseguito anche se la predizione indicava il contrario
- Quando rientriamo nel loop alla fine della prima interazione la nostra predizione indica che dovremmo uscire (ultima interazione del loop precedente) mentre in realtà effettueremo il salto.

Per ovviare a questo problema sfruttiamo una BHT a due bit nella quale sono necessarie due predizioni sbagliate consecutive per cambiare la nostra predizione. Per ogni indice della tabella i due bit sono utilizzati per indicare un dei quattro stati della macchina a stati finiti in Figura 35. Tale tecnica si può generalizzare fino ad utilizzare una tabella con  $n$  bit per ogni record. Il valore che può assumere questo record va da 0 a  $2^n - 1$ ; quando il valore del record diventa uguale ad almeno la metà del suo valore massimo ( $2^{n-1}$ ) la predizione del salto indica che esso deve essere eseguito, altrimenti la predizione sarà che non deve essere eseguito. Nello schema precedente il contatore veniva incrementato quando il salto veniva intrapreso e decrementato quando non veniva intrapreso.

Tuttavia anche se una generalizzazione è possibile gli studi hanno dimostrato che una tabella

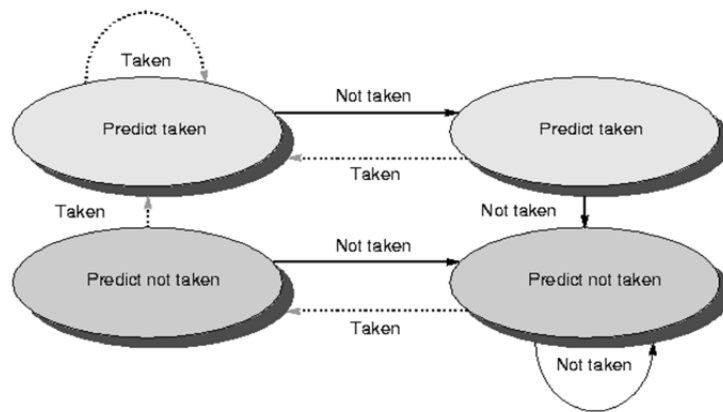


Figura 35: Macchina a stati finita per una BHT a 2 bit

a 2 bit fornisce dati più che soddisfacenti. Ad esempio per una architettura IBM *SPEC89* con una BHT con 4K record di 2 bit l'accuratezza nella predizione varia dal 99% all'82%.

La BHT a 2 bit utilizza solo i risultati delle esecuzioni precedenti di un singolo salto per predire il risultato di quel salto. L'idea di base però è che il comportamento dei salti recenti è che essi sono correlati con il comportamento di altri salti e perciò la predizione può essere influenzata da tali comportamenti. Un esempio è mostrato in Figura 36 Un predittore che utilizza il comportamento

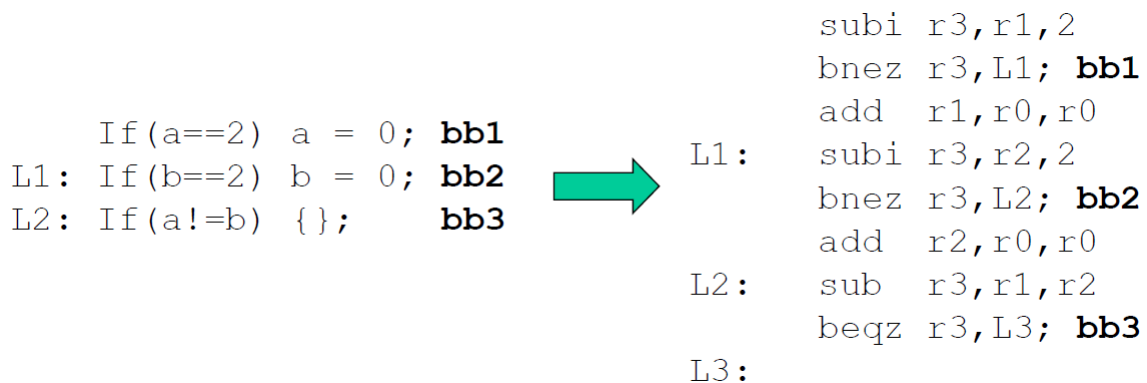


Figura 36: Esempio di salti correlati

di altri salti per effettuare una predizione è chiamato **Correlating Predictors** o anche **2-Level Predictors**. Un esempio di un *(1,1) Correlating Predictors* è un predittore ad un bit con un bit di correlazione, ovvero il comportamento dell'ultimo salto è utilizzato per scegliere una coppia di predittori ad un bit come mostrato in Figura 37. Si registrano le esecuzioni degli ultimi k salti; la predizione si basa sull'esecuzione del salto precedente selezionando la BHT ad un bit appropriata:

- Una predizione è usata nel caso in cui l'ultimo salto è stato eseguito.
- L'altra predizione è utilizzata se l'ultimo salto non è stato intrapreso.

In generale l'esecuzione dell'ultimo salto non riguarda la stessa istruzione sulla quale si cerca di fare una predizione come normalmente accade nei loop semplici.

In generale possiamo costruire un predittore correlato con (m,n) dove *m* indica gli ultimi *m*

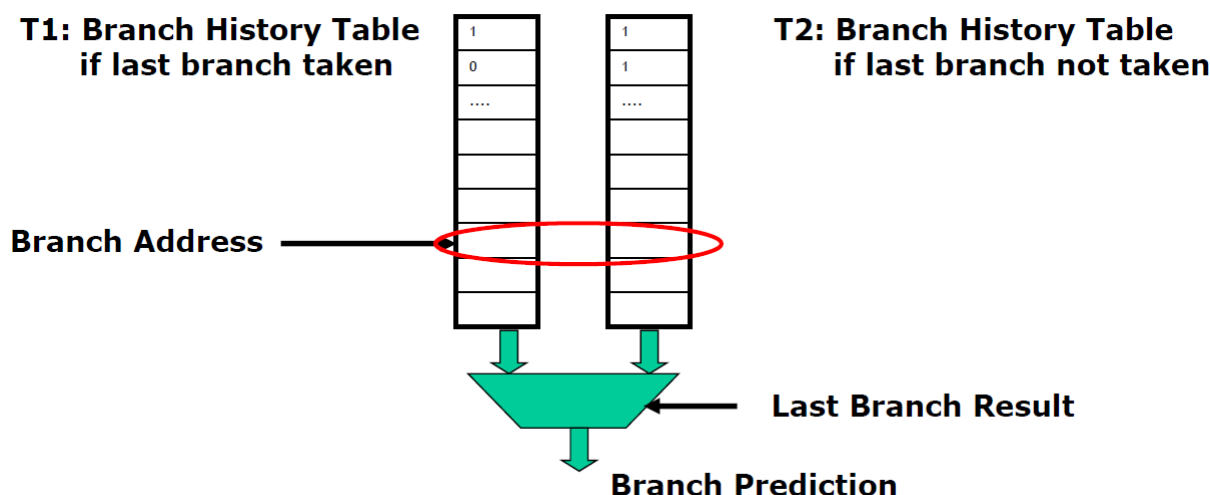


Figura 37: Esempio di predittore correlato di tipo (1,1)

salti da analizzare selezionando  $2^m$  BHT ognuna delle quali è un predittore a  $n$  bit. Il branch prediction buffer può essere indicizzato concatenando la parte finale del branch address con gli  $m$  bit della *global history*. Un esempio di un predittore correlato è un predittore (2,2) dove si hanno quattro BHT a 2 bit tra i quali scegliere e si usano 2 bit dalla global history per selezionare quale utilizzare come mostrato in Figura 38. Un predittore a due bit non correlato

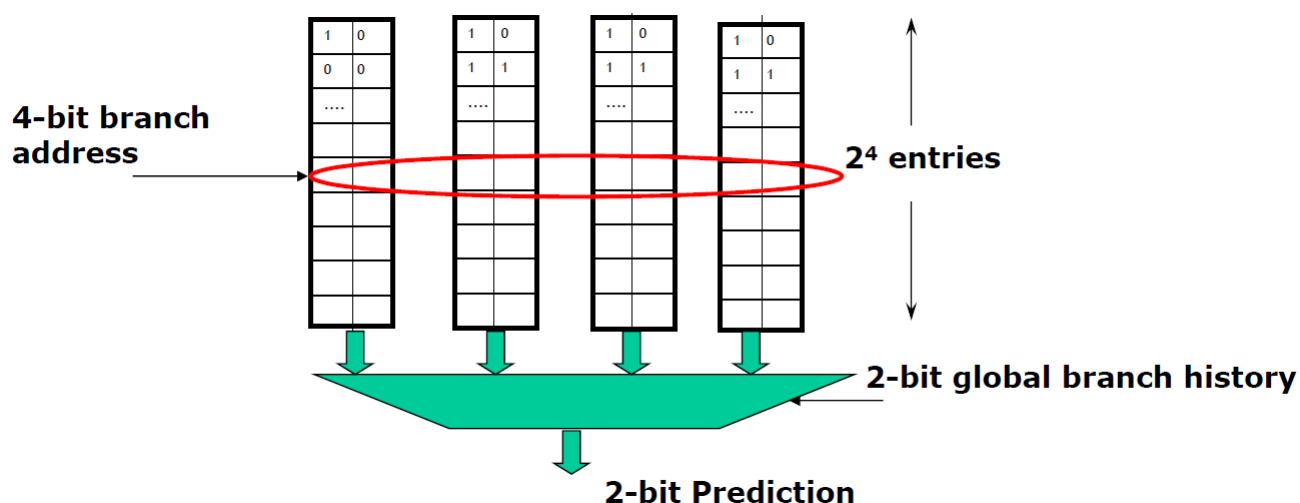


Figura 38: Esempio di predittore correlato (2,2)

non è altro che un predittore correlato con i valori (0,2); a questo punto possiamo confrontare le performance nel caso di un predittore semplice a 2 bit con una tabella di 4K entità e un predittore correlato (2,2) con tabelle di 1K entità. Come vediamo dal grafico in Figura 39 Il predittore correlato è, in molti casi più efficace di un predittore semplice mentre nei casi peggiori ne eguaglia le performance. Un'altra tecnica di predizione dinamica è quella del **Predittore di salto adattativo a due livelli** (*Two-Level Adaptive Branch Predictors*) nel quale un primo livello di storia viene memorizzato in uno shift register a  $k$  bit chiamato **Branch History Register (BHR)** il quale memorizza i risultati degli ultimi  $k$  salti. Il secondo livello di storicizzazione è

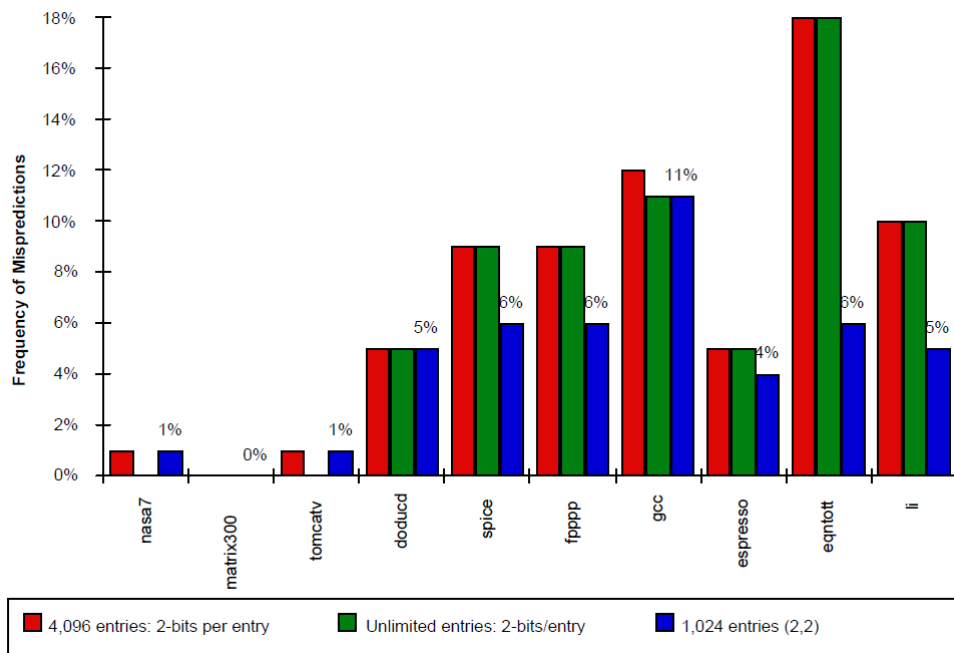


Figura 39: Comparazione delle performance per predittore non correlato e predittore correlato

memorizzato in una tabella con record di due bit chiamata **Pattern History Table (PHT)** per indicare la predizione. La BHR è utilizzata per indicizzare la PHT e selezionare i due bit da utilizzare; per selezionare quale dei due bit utilizzare si utilizza lo stesso principio utilizzato per i predittori a due bit semplici. Un evoluzione di questo predittore è il **predittore GShare** dove le informazioni dell'indirizzo locali vengono correlate con quelle globali tramite un'operazione di XOR. Un ultimo elemento importante nella predizione dinamica è il **Branch Target Buffer** la quale è una cache nella quale vengono memorizzate l'indirizzo di destinazione del salto per le istruzioni dopo il salto. Accediamo al BTB nello stage IF utilizzando l'indirizzo dell'istruzione da prelevare per indicizzare la cache. Un possibile esempio di record è mostrato in Figura 42. L'indirizzo di destinazione del salto è espresso sempre in modo relativo al PC. La struttura del BTB è mostrata in Figura 43; in tale buffer dobbiamo memorizzare solo gli indirizzi per quei salti che vengono eseguiti.

## 2.3 Speculazione

Senza tecniche di predizione di salto il parallelismo risulta molto limitato e si riduce all'analisi dei **basic block** ovvero a pezzi di codice nei quali non entrano o escono dei salti. Tuttavia possiamo azzardare alcune supposizioni di parallelismo tra diversi blocchi base effettuando delle speculazioni. Tramite le speculazioni possiamo recuperare ed eseguire le istruzioni come se le nostre predizioni siano corrette gestendo in seguito il caso in cui non siano corrette. Tale speculazione può essere supportata sia dal compilatore che dall'hardware.

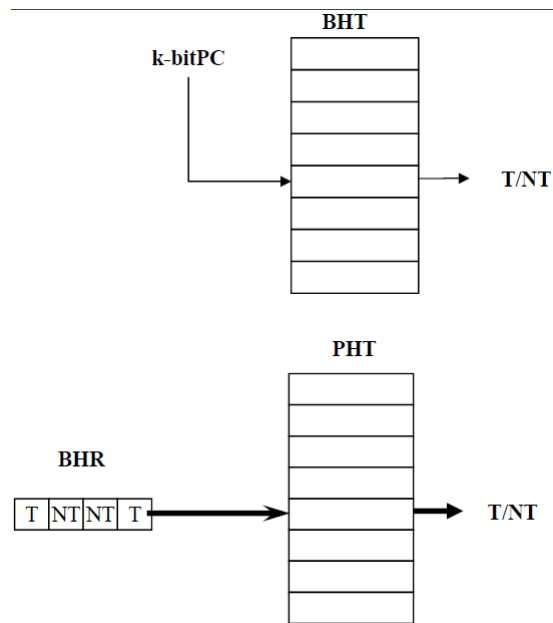


Figura 40: Esempio di predittore adattativo

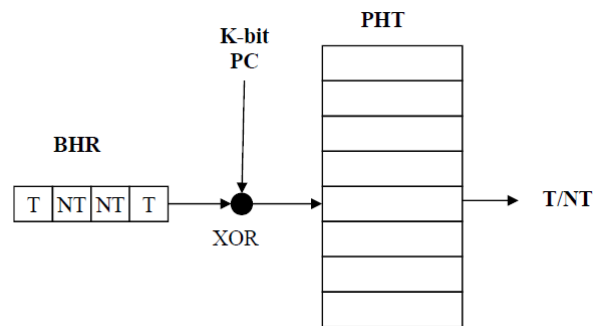


Figura 41: Esempio di predittore GShare

Exact Address of a Branch	Predicted target address
---------------------------	--------------------------

Figura 42: Esempio di record di un Branch Target Buffer



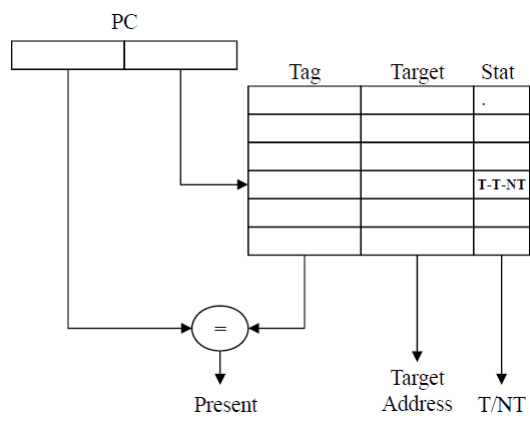


Figura 43: Struttura di un Branch Target Buffer

### 3 Instruction Level Parallelism

Come abbiamo visto nei capitoli precedenti in una macchina fornita di pipeline possiamo dimostrare come il numero di clock necessari per eseguire un'istruzione sia:

$$CPI_{pipeline} = CPI_{ideale} + Stalli\ Strutturali + Stalli\ Data\ Hazard + Stalli\ di\ Controllo + Stalli\ di\ Memoria$$

La riduzione di uno qualsiasi dei termini sulla destra da in modo che  $CPI_{pipeline}$  si avvicini sempre più al  $CPI_{ideale}$ . Il caso migliore si ha quando il throughput è massimo ed è uguale a 1

$$IPC_{ideale} = 1; CPI_{ideale} = 1$$

Tuttavia si hanno dei limiti alle performance dovuti ai diversi tipi di *rischi* (*Hazards*), questi possono essere di diversa natura:

- **Strutturali:** si possono risolvere tramite l'introduzione di nuovo hardware.
- **Dati:** necessitano di *forwarding* e di una schedulazione del codice a livello di compilazione.
- **Controllo:** Early evaluation, Branch Delay Slot, Predizione dei salti statica e dinamica.

Inoltre per le pipeline più profonde (superpipelining) questi problemi sono accentuati. In questo capitolo ci concentreremo su come incrementare il  $CPI$  oltre il valore ideale; per fare ciò però dobbiamo prima capire quali sono gli *hazard* sui dati che possiamo incontrare.

#### 3.1 Tipi di Hazards sui dati

Gli hazard innanzitutto sono quei conflitti che avvengono a tempo di esecuzioni e sono generati da dipendenze a livello di istruzione. Consideriamo l'esecuzione di un'istruzione generale di questo tipo:

$$r_k \leftarrow (r_i) \text{ op } (r_j)$$

Possiamo avere tre tipi di dipendenza a livello di istruzione come mostrato in Figura 44

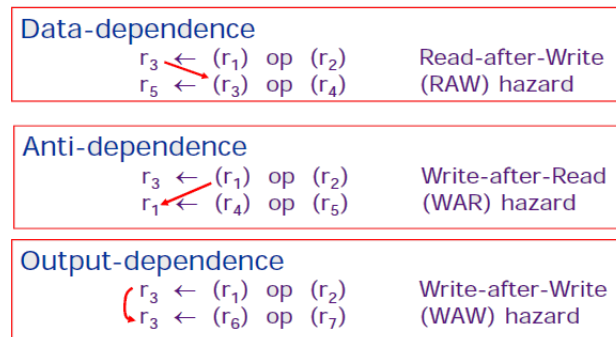


Figura 44: Esempi di dipendenza dei dati

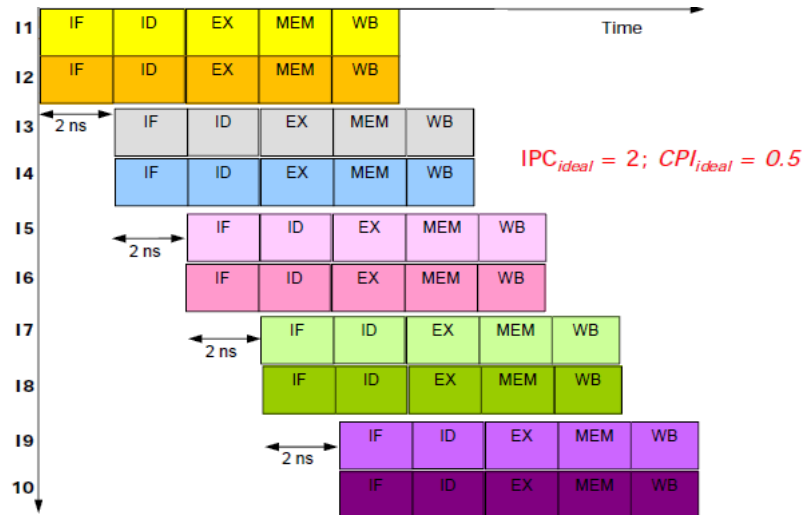


Figura 45: Esecuzione di istruzione in una pipeline dual-issue

### 3.2 Parallelismo a livello di istruzione

Per raggiungere livelli di performance maggiori è necessario estrarre dai programmi maggiore parallelismo, questo si traduce in pipeline con più uscite (*multiple-issue*). Per fare ciò è necessario individuare e risolvere le dipendenze, inoltre è utile riordinare (*schedule*) le istruzioni in modo da ottenere un maggiore parallelismo a tempo di esecuzione compatibilmente con le risorse a disposizione.

Per dare una definizione formale del parallelismo a livello di istruzione (*ILP*) possiamo dire che

*ILP = Sfruttare la potenziale esecuzione sovrapposta di istruzione non correlate*

Tale sovrapposizione è possibile tutte le volte che:

- Non abbiamo degli *Hazard* di tipo strutturale
- Non abbiamo *Hazard* di tipo RAW, WAR oppure WAW
- Non abbiamo *Hazard* di controllo

Un esempio di sistema *dual-issue* è mostrato in Figura 45 e Figura 46. In pipeline di tipo *multiple-issue* il  $CPI_{ideale}$  risulta essere  $< 1$  ad esempio considerando il caso ottimale di un processore *2-issue* abbiamo che ad ogni ciclo di clock vengono completate 2 istruzioni questo significa che:

$$IPC_{ideale} = 2; CPI_{ideale} = 0.5$$

Uno degli aspetti più critici nel caso di *ILP* è la determinazione delle dipendenze tra le istruzioni; infatti, se due istruzioni sono dipendenti tra loro esse non possono essere eseguite in parallelo ma dovranno essere eseguite in sequenza o al più in parziale sovrapposizione. Possiamo distinguere tre tipi di dipendenza:

- Dipendenza dei dati (Vera dipendenza)
- Dipendenza dei nomi
- Dipendenza di controllo

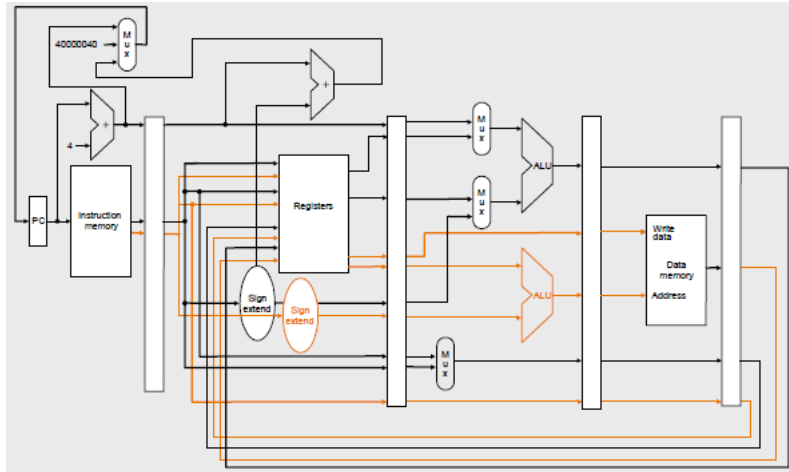


Figura 46: Schema hardware per una pipeline dual-issue con una unità ALU/BR e una unità load/store

**Dipendenza dei nomi** Tale dipendenza avviene quando due istruzioni usano lo stesso registro o la stessa area di memoria (chiamata *nome*) ma non esiste alcun flusso di dati tra queste due istruzioni. Possiamo individuare due tipi di dipendenza dai nomi tra due istruzioni *i* e *j* nelle quali *i* precede *j*

- **Antidipendenza:** quando l'istruzione *j* scrive un registro che l'istruzione *i* legge; l'ordine originale delle istruzioni deve essere preservato per essere sicuri che *i* legga il valore corretto (WAR).
- **Output dipendenza:** quando *i* e *j* scrivono lo stesso registro o la stessa area di memoria; l'ordine delle istruzioni deve essere rispettato per essere sicuri che il valore finale sia quello scritto da *j*

In realtà la dipendenza dai nomi non è una vera e propria dipendenza in quanto non vi è alcuno scambio di valori tra le istruzioni; se il *nome* usato nell'istruzione può essere cambiato non ci sono conflitti. Tuttavia per quanto riguarda le aree di memoria è molto più difficile localizzare questo tipo di conflitto infatti due indirizzi possono essere diversi ma puntare alla stessa area (*memory disambiguation*) mentre una rinominazione dei registri risulta molto più semplice. La rinominazione può essere effettuata sia staticamente dal compilatore sia in modo dinamico dall'hardware.

**Dipendenze dei dati** Le dipendenze dei dati possono potenzialmente generare dei *Data Hazard* ma l'impatto che questi hazard hanno in termini di stalli e tecniche di eliminazione degli stalli sono caratteristiche specifiche della pipeline e non dipendono dalla dipendenza. Le *RAW* sono le uniche vere dipendenze sui dati che abbiamo. Le dipendenze sono una caratteristica del programma mentre gli *hazard* sono specifiche della pipeline.

**Dipendenze di controllo** Le dipendenze di controllo sono determinate dall'ordinamento delle istruzioni e sono preservate da due proprietà:

- Le istruzioni devono essere eseguite nell'ordine del programma per assicurare che un'istruzione che si trova prima di un salto venga eseguita prima del salto.

- Individuazione degli *hazard di controllo* per assicurare che un'istruzione che dipende da un salto non sia eseguita prima di conoscere la direzione del salto.

Sebbene preservare il controllo delle dipendenze sia un modo semplice per preservare l'ordine del programma esso non è così essenziale da dover essere preservato.

### 3.2.1 ILP in pratica

Dalla trattazione appena fatta possiamo ricavare due proprietà importanti per verificare la correttezza di un programma (e che in realtà preservano sia le dipendenze dei dati che quelle di controllo):

- **Data flow:** il flusso dei valori dei dati attraverso le istruzioni deve produrre il risultato corretto.
- **Exception behavior:** preservare il comportamento delle eccezioni che significa che qualsiasi cambiamento nell'ordine di esecuzione delle istruzioni non deve cambiare come le eccezioni sono sollevate dal programma.

Esistono due tecniche fondamentali per supportare e implementare l'*ILP*, lo **scheduling dinamico** che dipende dall'hardware per localizzare il parallelismo e lo **scheduling statico** che fa affidamento sul software per individuare possibili parallelismi. La prima soluzione è quella più utilizzata in ambito desktop e server.

Consideriamo ora un processore di tipo *single-issue* lo stage IF precede quello EXE e le istruzioni possono essere prelevate sia dall'*Instruction Register* sia da una coda di istruzioni pendenti. La fase di esecuzione può richiedere più cicli di clock in base al tipo di operazione.

**Scheduling dinamico** Il problema principale è quello che non si può nascondere una dipendenza dai dati senza causare uno stallo nell'esecuzione della pipeline. Una soluzione è quella di permettere alle istruzioni situate dopo lo stallo di procedere; l'hardware deve riordinare dinamicamente l'esecuzione delle istruzioni per dar in modo di ridurre gli stalli. Per fare ciò è necessario permettere un'esecuzione fuori ordine e una fase di commit finale.

L'hardware riordina l'esecuzione delle istruzioni per ridurre il numero degli stalli della pipeline mentre mantiene il *data flow* e *exception behavior*. I vantaggi principali di questa tecnica sono il fatto di permettere una gestione di alcuni casi in cui esistono delle dipendenze non note al tempo della compilazione, inoltre permette di semplificare la complessità del compilatore ed infine permette di compilare il codice affinché esso venga eseguito efficientemente anche su pipeline diverse. Questi vantaggi tuttavia hanno un costo che comporta un significativo incremento della complessità dell'hardware, un incremento dei consumi e può generare delle eccezioni imprecise. Tale soluzione è utilizzata soprattutto nei *Processori Super-scalari*

**Scheduling statico** In questo caso il compilatore utilizza dei sofisticati algoritmi per individuare e organizzare il codice in modo da sfruttare l'*ILP*, per fare ciò analizza i **basic block** e ricerca il parallelismo in questi seppur esso sia minimo (15% - 25%), ed inoltre sfrutta il parallelismo tra diversi *basic block*. Un limite a questa tecnica è però la dipendenza dai dati presente nei vari blocchi base. Tipicamente questa tecnica è sfruttata dai processori **VLIW** (*Very Long Instruction Word*) i quali si aspettano del codice privo di dipendenze dal compilatore.

Il limite principale di questa tecnica è dato dall'imprevedibilità dei salti, dalla latenza della memoria, dalla dimensione del codice e dalla complessità del compilatore.

### 3.2.2 Esecuzione super-scalare e VLIW

Passando al caso *multi-issue* la questione si complica anche se le prestazioni migliorano come vediamo in Figura 47, infatti si vogliono eseguire più istruzioni per ogni ciclo di clock; per fare ciò è necessario prelevare più istruzioni per ciclo dall'IR e questo dipende dalla banda a disposizione. La cosa più difficile però risulta essere l'analisi delle dipendenze dei dati e di controllo per le istruzioni da eseguire. In Figura 48 vediamo come è strutturato un processore super-scalare nel

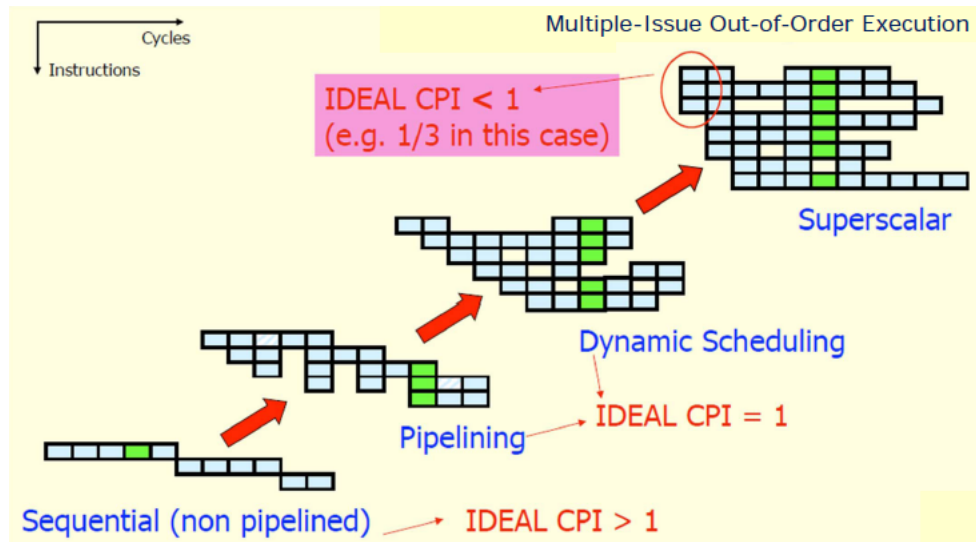


Figura 47: Confronto di prestazioni tra architetture

quale possiamo notare le diverse unità per l'esecuzione di istruzioni parallele come le due ALU o l'unità per le istruzioni di load/store, e l'unità per il riordino delle istruzioni.

Per decidere quali istruzioni mandare in esecuzione si utilizza lo *Scheduler dinamico* il quale per

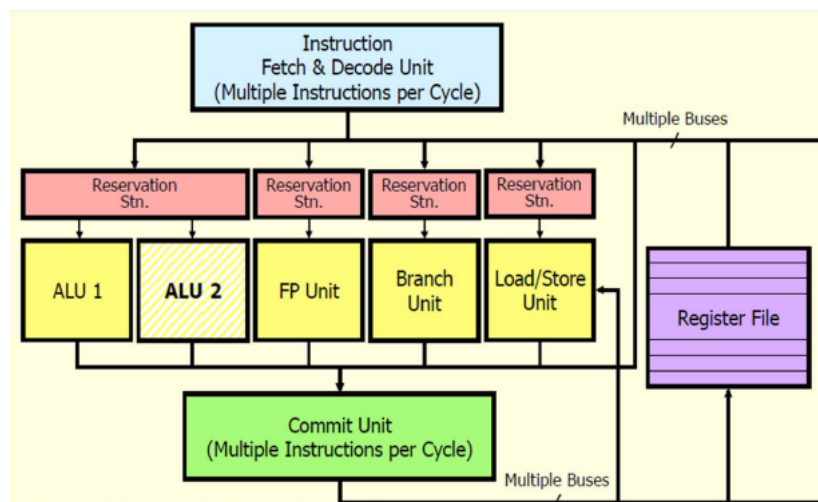


Figura 48: Struttura di un processore superscalare

ogni ciclo di clock analizza quali sono le dipendenze e per fare ciò la sua complessità è molto alta, nell'ordine del quadrato delle possibili istruzioni come mostrato in Figura 49. Esiste un limite al numero di istruzioni che possono essere analizzate durante un singolo ciclo di clock, infatti



Figura 49: Tabella delle dipendenze in uno scheduler dinamico

i limiti principali dei processori super-scalari riguardano tutti lo scheduler dinamico in quanto esso è molto costoso in termini di area in quanto la sua logica è molto complessa, il tempo di clock dipende dal tempo di analisi delle istruzioni ed infine la verifica del design dello scheduler è molto complessa. Queste limitazioni portano a delle limitazioni in termini di istruzioni che possono essere eseguite simultaneamente. Attualmente in realtà esistono dei processori a 6-issue anche se molto rari, più normalmente sono di tipo 4-issue o minori in quanto è troppo difficile trovare 8 o addirittura 16 istruzioni indipendenti in un singolo ciclo.

La famiglia di processori multi-issue che sfruttano lo scheduler statico sono, invece, i processori VLIW (*Very Long Instruction Word*) nei quali è il compilatore che decide cosa far fare ad ogni istruzione per ogni ciclo di clock come si vede in Figura 50. I processori di tipo super-scalare

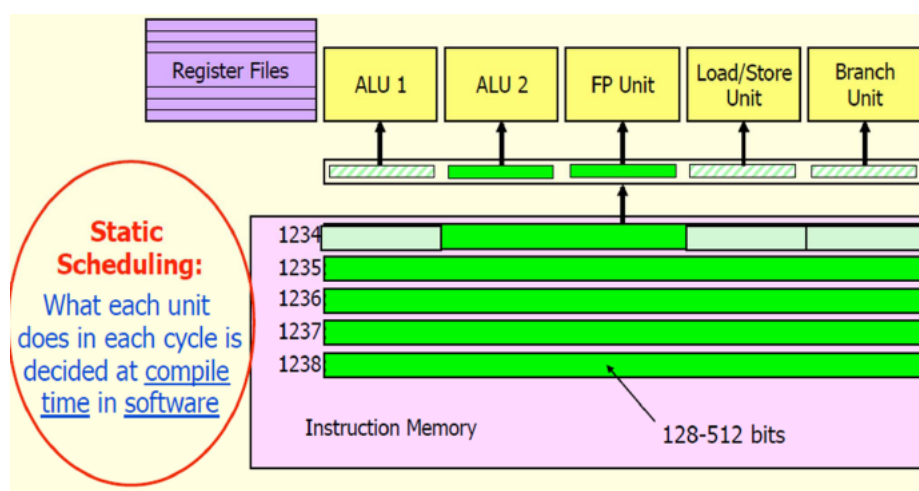


Figura 50: Scheduler statico nel caso di processori VLIW

sono utilizzati soprattutto in ambito desktop e server, mentre i processori VLIW sono utilizzati soprattutto in ambito embedded in quanto la decisione sull'esecuzione è presa a compile-time e non è necessario aggiungere dell'area per lo scheduler dinamico.

### 3.3 Scoreboard

Come abbiamo visto fino ad ora lo scheduling dinamico è il meccanismo più utilizzato nei sistemi general purpose per sfruttare il parallelismo tra le istruzioni. Tale meccanismo però ha diverse tecniche di implementazione, una di queste è lo **Scoreboard**. Lo *Scoreboard* divide il normale stage ID in due stage per permettere l'esecuzione delle istruzioni nell'ordine più efficiente. Questi due stage sono:

- Lo stage *issue* che si occupa di decodificare le istruzioni e di verificare eventuali hazard strutturali.
- Lo stage *read operands* (RR) che si occupa di risolvere i data hazard ritardando eventualmente la lettura dei registri.

Grazie a questo meccanismo le istruzioni vengono eseguite quando non hanno alcuna dipendenza o non esistono degli hazard strutturali non verifica però una eventuale priorità delle istruzioni. Per spiegare la struttura dello *Scoreboard* dobbiamo innanzitutto definire quando un'istruzione si dice in **esecuzione**. Possiamo distinguere quando un'istruzione inizia la sua esecuzione e quando essa la termina durante questi due istanti l'istruzione si dice in *esecuzione*. In una pipeline possono esistere molte istruzioni in esecuzione nello stesso momento, questo richiede che esistano più unità funzionali. Quello che noi analizzeremo è la struttura di un **CDC 6600** nel quale le istruzioni vengono immesse in ordine ma vengono eseguite e completate in un ordine casuale. L'architettura di questo sistema è mostrata in Figura 51 Nella pipeline di uno scoreboard

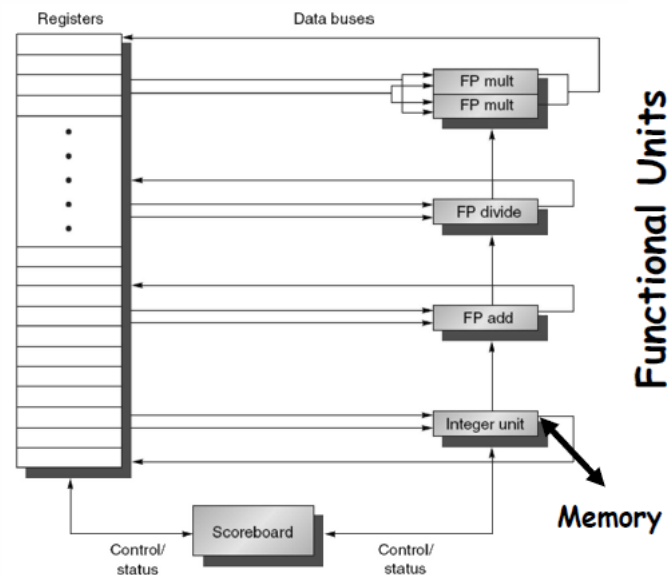


Figura 51: Architettura di un sistema *Scoreboard*

le fasi ID, EXE e WB sono sostituite da quattro stage. Lo stage ID è diviso in due parti, la prima chiamata **Issue** la quale decodifica l'istruzione e controlla eventuali hazard strutturali, la seconda chiamata **Read Operands** che attende fino alla risoluzione degli hazard sui dati. Lo scoreboard fa in modo che le istruzioni eseguite siano prive di dipendenze. Come abbiamo visto le istruzioni vengono prelevate in ordine dallo stage *Issue* ma da quell'istante esse possono essere ordinate in qualsiasi modo, infatti durante lo stage *read operands* esse possono essere bloccate o bypassate, inoltre, la latenza di esecuzione può variare tra le diverse unità funzionali. Un completamento fuori ordine può portare però a hazard di tipo WAR o WAW che tuttavia possono essere facilmente risolti infatti per i WAR è sufficiente:

- Bloccare il *write back* finché i registri non vengono letti
- Effettuare la lettura dei registri soltanto durante la fase di *Read Operands*

Mentre per risolvere i WAW è sufficiente individuare l'hazard e bloccare l'esecuzione delle istruzioni dipendenti successive fino a quando esse non vengono concluse.



L'individuazione e la risoluzione degli hazard è centralizzata nello *scoreboard*; ogni istruzione attraversa lo scoreboard dove viene aggiornata una tabella delle dipendenze, a questo punto lo scoreboard determina quando l'istruzione può leggere i registri ed iniziare la sua esecuzione. Se un'istruzione non può cominciare immediatamente la sua esecuzione lo scoreboard monitora ogni cambiamento e decide quando l'istruzione può andare in esecuzione. Infine, lo scoreboard, controlla quando l'istruzione scrive il risultato dentro i registri di destinazione.

Un problema che si viene a creare quando si accetta il completamento delle istruzioni fuori ordine è quello della preservazione del comportamento delle eccezioni; una soluzione è quella di assicurarsi che nessuna istruzione possa generare una eccezione finché il processore non conosce esattamente l'istruzione che ha sollevato l'eccezione. Un'eccezione si dice *imprecisa* se lo stato del processo nell'istante in cui viene sollevata una eccezione non è uguale a quello del caso in cui l'istruzione sia eseguita in ordine; un'eccezione imprecisa si può verificare quando la pipeline ha già completato delle istruzioni che sequenzialmente si trovano dopo l'istruzione che ha sollevato l'eccezione, oppure se non ha ancora completato istruzioni che sequenzialmente la precedono.

**I quattro stadi dello Scoreboard** Analizziamo ora in dettaglio i quattro stadi dello Scoreboard. Il primo stadio è quello di *Issue* nel quale le istruzioni vengono decodificate e si verificano gli eventuali hazard strutturali e quelli di WAW. Le istruzioni lasciano questo stage in ordine, se l'unità funzionale necessaria per eseguire l'istruzione è libera e non esistono altre istruzioni che hanno lo stesso registro di destinazione (no WAW) lo stage issue inoltra l'istruzione all'unità funzionale e aggiorna la sua struttura interna. In caso contrario lo stage ferma l'istruzione fino a quando tutti gli hazard sono stati risolti. Ogni qualvolta lo stage di issue ferma un'istruzione il buffer tra IF e Issue si riempie. Nel caso il buffer sia singolo IF si blocca e attende l'Issue nel caso invece il buffer sia a coda l'IF si blocca solo nel caso in cui la coda sia piena.

Lo stage *Read Operands* attende la risoluzione dei data hazard e legge i registri. Un registro risulta disponibile quando non ci sono istruzioni attive precedenti che scrivono su di esso oppure se un'unità funzionale scrive il suo risultato in tale registro. Quando i registri sono disponibili lo scoreboard comunica all'unità funzionale di leggere i registri. Gli RAW hazard vengono risolti dinamicamente in questa fase.

Lo stage *execution* è lo stage composto dalle unità funzionali; durante questo stage le unità funzionali svolgono le diverse operazioni, quando il risultato è pronto viene notificato allo scoreboard. Le unità funzionali sono caratterizzate da una latenza variabile, il tempo per iniziare l'esecuzione è variabile in quanto è il tempo necessario per leggere i diversi registri, i tempi per effettuare una load/store variano in base ai cache HIT/MISS.

L'ultimo stage è denominato *Write Result*, in questa fase si controllano eventuali hazard di tipo WAR e si termina l'esecuzione scrivendo il risultato nel registro di destinazione.

**Lo Scoreboard in pratica** Lo scoreboard è formato da tre unità fondamentali:

- Instruction status
- Functional Unit status: il quale indica lo stato delle unità fondamentali tramite diversi indici

**Busy:** indica se la FU è occupata oppure no

**Op:** indica l'operazione da eseguire

$F_i$ : registro di destinazione

$F_j, F_k$ : registri sorgenti

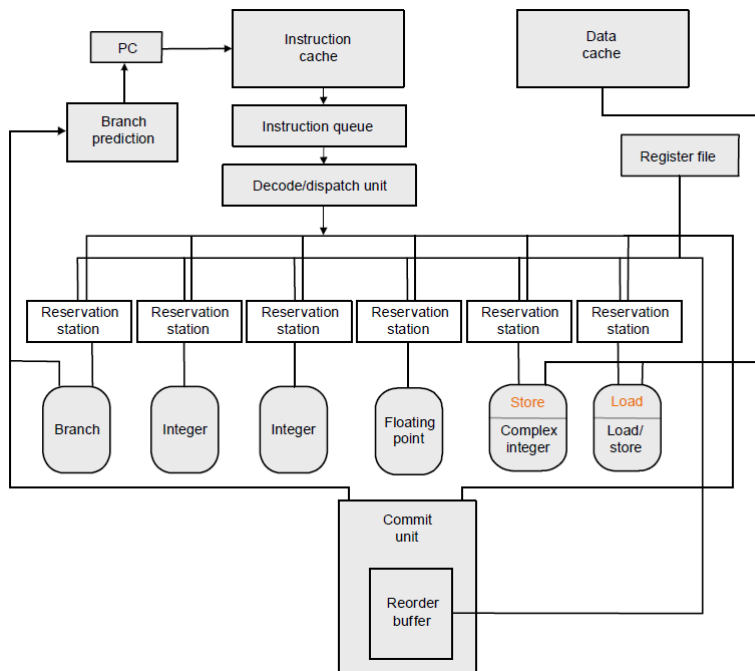


Figura 52: Architettura per l'algoritmo di Tomasulo

$Q_j, Q_k$ : indicano quale FU sta tenendo occupato il registro corrispondente

$R_j, R_k$ : sono dei flag che indicano lo stato dei registri sorgenti.

- Register Result status: indica quale FU dovrà scrivere sui registri di destinazione.

Per un esempio completo si rimanda alle slide dell'insegnante.

### 3.4 Algoritmo di Tomasulo

Un altro tipo di algoritmo da sfruttare per lo scheduling dinamico è l'algoritmo di *Tomasulo* introdotto da IBM tre anni dopo il CDC 6600 lo scopo di tale algoritmo è sempre quello di ottenere prestazioni elevate senza l'utilizzo di speciali compilatori.

A differenza dello Scoreboard tuttavia il meccanismo di controllo e di buffer è distribuito sulle diverse unità funzionali. I buffer associati alle unità funzionali sono chiamate *Reservation Station*. I registri nelle istruzioni sono sostituiti da valori o puntatori alle reservation station. È possibile così il renaming dei registri. In questo modo si evitano anche eventuali hazard di tipi WAR e WAW; inoltre esistono più *RS* che registri e questo permette delle ottimizzazioni che il compilatore non può fare. Infine il risultato delle FU non transita dai registri bensì viene diffuso a tutte le altre FU tramite un *Common Data Bus*. Le operazioni di Load e Store sono trattate come normali FU. L'architettura necessaria per implementare l'algoritmo di Tomasulo è mostrata in Figura 52 mentre la struttura di una unità funzionale è mostrata in Figura 53. I vari campi che compongono una reservation station sono:

**Tag:** identifica quale RS è coinvolta.

**Busy:** identifica se la RS è occupata.

**OP:** Identifica il tipo di operazione eseguita dal componente.

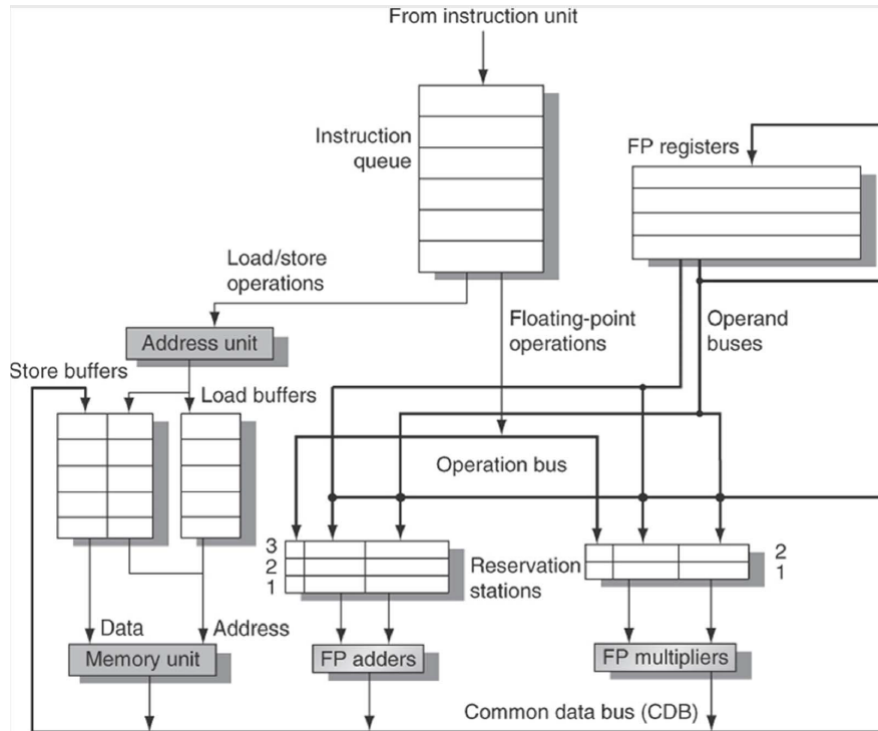


Figura 53: Architettura di un'unità funzionale

$V_j, V_k$ : Valori contenuti nei registri; per la *load*  $V_j$  contiene il valore dell'offset.

$Q_j, Q_k$ : Puntatori alle RS che producono i valori  $V_j, V_k$  se il valore è uguale a 0 l'operando è già disponibile.

Si noti che solo uno dei due campi  $V$  e  $Q$  è disponibile per ogni operando in un determinato istante.

Il register file e lo Store buffer hanno un campo *Value* ( $V$ ) e uno *Puntatore* ( $Q$ ) il quale punta al numero della *reservation station* che produce il risultato da immagazzinare, se questo puntatore è uguale a zero allora non ci sono istruzioni attive e il valore contenuto nel RF/Buffer è quello corretto. Nel caso di Load buffer abbiamo un campo *Address* ( $A$ ) e un campo *Busy*; il campo  $A$  mantiene le informazioni riguardanti gli indirizzi calcolati nelle operazioni di Load/Store, all'inizio contiene le informazioni sull'offset poi, una volta calcolato, contiene l'indirizzo effettivo.

### 3.4.1 Gli stadi dell'algoritmo di Tomasulo

**Il primo stadio: ISSUE** Durante questo stadio si preleva un'istruzione  $I$  dalla testa della coda delle istruzioni (**in-order issue**). Si controlla se la RS associata a quell'istruzione è vuota altrimenti si attende (controllo sugli structural hazard). Nel caso gli operandi non siano ancora disponibili si tiene traccia della FU che produce tali operandi (puntatore  $Q$ ). Durante questa fase si effettua anche il *Rename* dei registri in modo tale da evitare eventuali *WAR*, infatti, se un'istruzione  $I$  scrive un registro  $R_x$  e un'istruzione  $K$  già prelevata legge tale registro  $K$  conosce già il valore di  $R_x$  e lo ha immagazzinato nella sua RS oppure conosce quale operazione produce tale valore. Inoltre si evitano anche eventuali *WAW* in quanto le istruzioni sono prelevate in ordine.

**Secondo stadio: Esecuzione** Quando entrambi gli operandi sono disponibili si esegue l'operazione; nel caso in cui non siano pronti invece si controlla il *Common Data Bus* in attesa del risultato; ritardando l'esecuzione si evitano eventuali RAW. Si noti come più istruzioni possono diventare eseguibili allo stesso istante per la stessa FU a questo punto bisogna verificare la disponibilità dell'unità funzionale. Le RAW hazard sono molto meno incisive in quanto sono gestite a livello di RS e non è necessario attendere il *write back* sul register file.

Per quanto riguarda le istruzioni di load e di store l'esecuzione avviene in due passi, nel primo passo si calcola l'effettivo indirizzo di destinazione e si memorizza nel buffer, nel secondo passo in caso di load si esegue l'operazione non appena l'unità è disponibile nel caso della store si attende invece che il dato da immagazzinare sia disponibile.

Per preservare il comportamento dell'esecuzione nessuna istruzione può cominciare l'esecuzione fino a quando i branch precedenti non sono stati eseguiti. Se si usano tecniche di predizione la CPU deve conoscere se la predizione è corretta prima di procedere.

**Terzo stadio: Write result** Quando un risultato è disponibile esso viene scritto sul *Common Data Bus* e da qui copiato sia sul RF sia su tutte le RS che attendono questo risultato. Il *Common Data Bus* è un bus di tipo data+source composto da 64 bit di dati e 4 bit per la sorgente in questo modo le FU possono effettuare un lookup associativo.

### 3.4.2 Alcuni dettagli

Le operazioni di load e di store attraversano un'unità funzionale per il calcolo dell'indirizzo effettivo prima di procedere con le vere e proprie operazioni di load e di store. La load necessita di una seconda fase per accedere alla memoria mentre invia il risultato al RF e alle RS durante lo stage *Write Result*. La store, invece completa la sua esecuzione durante lo stage *Write Result* nel quale scrive i dati sul buffer. Le operazioni di load e di store possono essere eseguite in ordine differente purché esse accedano a differenti aree di memoria, in caso contrario possono presentarsi problemi di WAR (scambio tra load e store), di RAW (scambio tra store e load) oppure di WAW (scambio tra due store) invece le load possono essere scambiate liberamente. Per identificare questo tipo di anomalie il calcolo degli indirizzi di tutte le operazioni deve essere calcolato dalla CPU e quindi secondo l'ordine del programma.

Prendiamo il caso di una load eseguita fuori ordine con una store precedente ed assumiamo che il calcolo dell'indirizzo sia eseguito con l'ordine del programma. Quando l'indirizzo della load è calcolato esso viene comparato con i campi *A* dello *Store Buffer* nel caso vi sia un match la load non viene inviata allo Load Buffer fino a quando il conflitto non è risolto. Le operazioni di store invece verificano la presenza di conflitti sia nello Store Buffer che nel Load Buffer.

### 3.4.3 Tomasulo in pratica

Per un esempio completo si rimanda alle slide dell'insegnante per il funzionamento dell'algoritmo.

### 3.4.4 Tomasulo vs Scoreboard

Al contrario dello scoreboard l'algoritmo di Tomasulo ha una finestra di prelevamento delle istruzioni minore (5 vs 12) in entrambi i casi non si hanno hazard di tipo strutturale nel prelevamento delle istruzioni, nel caso di Tomasulo questi sono bloccati a livello di RS mentre nel caso dello Scoreboard a livello di FU. Tomasulo è più efficiente per quanto riguarda la risoluzione di WAW e di WAR che vengono risolti tramite renaming mentre per lo Scoreboard sono necessari alcuni stalli; inoltre, in Tomasulo il risultato di una FU è distribuito a tutte le altre tramite

il Common Data Bus mentre nello scoreboard è necessario attendere che il risultato sia scritto nei registri di destinazione. Il controllo in Tomasulo è distribuito ed è possibile effettuare un loop unrolling al contrario dello Scoreboard. Tuttavia i limiti di Tomasulo risiedono nella sua complessità e alla limitazione delle prestazioni del Common Data Bus; inoltre il parallelismo è ridotto a causa dei salti.

### 3.5 Register Renaming

#### 3.5.1 Renaming implicito

Come abbiamo visto nel paragrafo precedente l'algoritmo di Tomasulo fornisce il *register renaming* in modo implicito tramite le *reservation station* questo permette a iterazioni diverse di utilizzare registri fisici diversi (**dynamic loop unrolling**), permette di sostituire i registri statici con dei puntatori dinamici che permettono in pratica di incrementare la dimensione del register file. Questo permette alle istruzioni di procedere e, tramite l'uso della branch prediction di prelevare più istruzioni di iterazioni diverse.

Per un esempio di questo meccanismo si rimanda all'esempio sulle slide della professoressa.

#### 3.5.2 Renaming esplicito

Come abbiamo appena visto tomasulo fornisce il renaming in modo implicito tramite l'utilizzo delle reservation station. Quello che vogliamo fare ora è capire come implementare un renaming dei registri in modo esplicito. Per fare questo innanzitutto dobbiamo tener conto che dobbiamo utilizzare un maggior numero di registri fisici rispetto a quelli specificati dall'ISA. Il principio chiave è quello di allocare una nuova destinazione fisica per ogni istruzione che scrive un risultato. Questa tecnica è molto simile alla trasformazione effettuata dal compilatore chiamata *Static Single Assignment (SSA)* ma in questo caso viene effettuata dallo hardware. Con questa tecnica si rimuovono tutte le possibilità di avere WAR o WAW. In Figura 54 vediamo come il *Register File Fisico* sia molto più grande di quello standard, inoltre, notiamo la presenza di una tabella denominata *Freelist* nella quale sono memorizzati i registri fisici che non sono utilizzati e che sono quindi *liberi*. Per implementare questo meccanismo è sufficiente tenere traccia dell'associazione dei registri tramite una *tabella di traduzione* come mostrato in Figura 55.

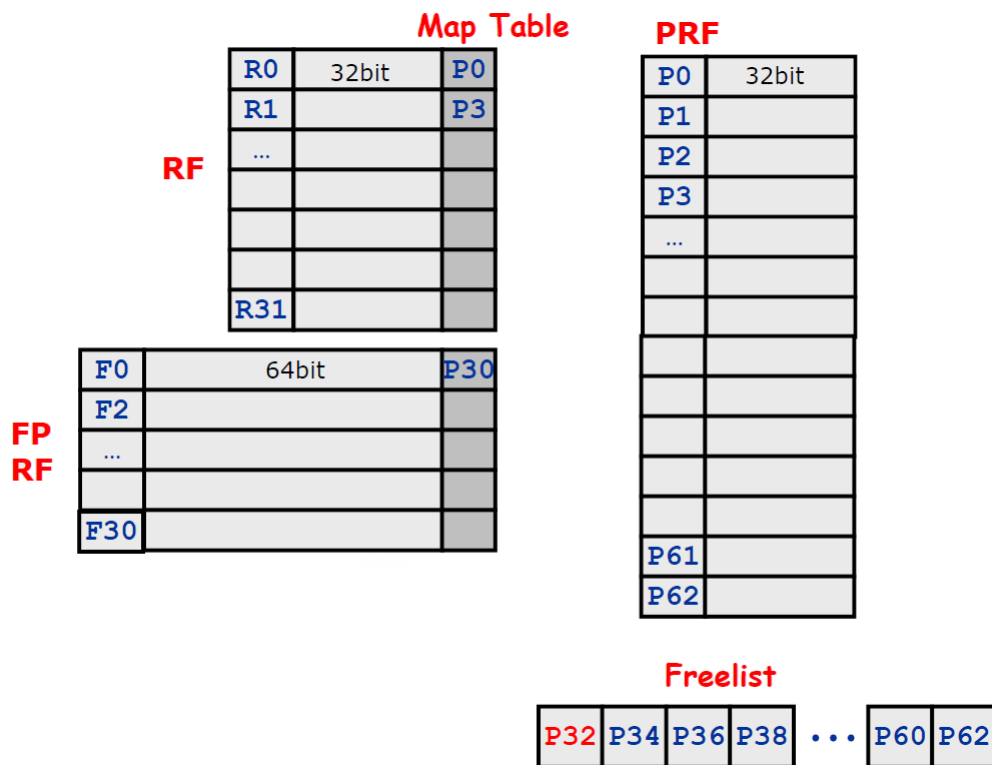


Figura 54: Associazione tra RF e RF fisico

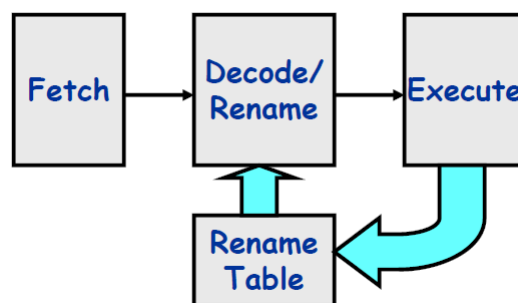


Figura 55: Meccanismo di register renaming