

Architetture Avanzate dei Calcolatori*

Daniele Iamartino
Stefano Cherubin

Indice

I Pipeline	6
1 Caratteristiche della architettura MIPS	6
1.1 Instruction Set Architecture (base) del processore MIPS	6
1.2 Formato delle istruzioni MIPS a 32 bit	6
1.2.1 Formato R per istruzioni ALU Register-Register	7
1.2.2 Formato I per istruzioni LOAD/STORE	7
1.2.3 Formato I per istruzioni immediate ALU	7
1.2.4 Formato I per istruzioni di branch condizionale	7
1.2.5 Formato J per i Jump incondizionati	7
1.3 Esecuzione delle istruzioni MIPS	8
1.4 Dettagli tecnici delle istruzioni MIPS	8
1.4.1 Latenza delle istruzioni	8
1.4.2 Operazioni MIPS riassunte fase per fase	8
1.4.3 Schema delle implementazioni	9
1.4.4 Implementazione multi-cycle	11
1.5 Pipelining	11
1.5.1 Esecuzione sequenziale vs. pipelining	11
1.5.2 Le varie fasi della pipeline riviste per i vari tipi di istruzione	12
1.5.3 Implementazione della pipeline MIPS	12
1.6 Hazards (pericoli dovuti a dipendenze)	12
1.6.1 Il problema	12
1.6.2 Structural Hazards	13
1.6.3 Data Hazards: introduzione	13
1.6.4 Data Hazards: Possibili soluzioni	13
1.6.5 Forwarding	14
1.6.6 Ottimizzazione della pipeline	16
1.6.7 Data Hazards nella pipeline ottimizzata	16
1.6.8 Classificazione dei Data Hazards (RAW - WAW - WAR)	17
1.7 Problemi di performance nella Pipeline	17
1.7.1 Metriche base di Performance	18
1.7.2 Altre metriche di performance	18
1.7.3 Metriche di performance asintotiche	18
1.7.4 Speedup della pipeline	19
2 Tecniche di predizione dei branch	20
2.1 Istruzioni di branch condizionali	20
2.2 Il problema dei control hazards	21
2.3 Soluzioni al problema del control hazard	21
2.3.1 Early evaluation del PC	22
2.4 Tecniche di predizione dei branch	23
2.4.1 Tecniche di predizione statica	23
2.4.2 Tecniche di predizione dinamica dei branch	26
II ILP	31
3 Instruction Level Parallelism	31
3.1 Portare il parallelismo a livello di istruzione	31
3.1.1 Riassunto sulla pipeline	31
3.1.2 Riassunto tipi di Data hazards	31
3.1.3 Evoluzioni della pipeline	32
3.1.4 Altri concetti base e definizioni	33
3.2 Approfondimenti sulle dipendenze	34
3.2.1 Name dependences	34
3.2.2 Data dependences e Hazards	34
3.2.3 Control dependences	34
3.2.4 Program Properties	34
3.3 Register Renaming	35

4 Evoluzione dello scheduling ILP: processori superscalari	35
4.1 Assunzioni	35
4.2 Dynamic Scheduling	35
4.2.1 Pro e Contro	36
4.2.2 Riassumendo	36
4.3 Static Scheduling	36
4.3.1 Evoluzione del parallelismo	37
4.4 Esecuzione superscalare	37
4.4.1 Struttura dello scheduler dinamico	38
4.4.2 Limiti	38
4.4.3 Riassunto di processori superscalari e scheduling dinamico	39
4.5 ReOrder Buffer (ROB)	39
5 Very Long Instruction Word (VLIW)	40
5.1 Idea di base	40
5.1.1 Come funziona	40
5.2 Caratteristiche	41
5.2.1 Limiti di VLIW	41
5.2.2 Vantaggi e svantaggi	41
5.2.3 Scheduling Software vs. Scheduling Hardware	42
5.2.4 Processori superscalari vs. VLIW	42
5.3 Scheduling VLIW	42
5.3.1 Grafo delle dipendenze	43
5.3.2 List based scheduling	43
5.3.3 Loop unrolling	43
5.3.4 Loop peeling & fusion	44
5.3.5 Software pipelining	44
5.3.6 Trace scheduling	45
5.3.7 Superblock scheduling	45
5.3.8 Predicated execution	46
6 Confronto architetture multiple issue	46
III Algoritmi e tecniche di dynamic scheduling	47
7 Punti cardine	47
7.1 Mantenere l'ordine	47
7.1.1 Issue	47
7.1.2 Execution	47
7.1.3 Commit	47
8 Scoreboard	47
8.1 Caratteristiche generali della scoreboard	47
8.1.1 Hazard e dipendenze	48
8.1.2 Gestione delle eccezioni:	48
8.2 I 4 stadi di controllo della scoreboard	49
8.3 Struttura della scoreboard	50
8.4 Esempio di funzionamento della scoreboard	50
8.5 Scoreboard con renaming esplicito	52
8.5.1 Variazioni rispetto alla versione base	53
9 Algoritmo di Tomasulo	53
9.1 Caratteristiche principali	53
9.2 Schema architetturale	54
9.2.1 Schema generale di una architettura	54
9.2.2 Schema di una Floating Point Unit	54
9.2.3 Componenti di una Reservation Station	54
9.3 Gli stadi dell'algoritmo	55
9.3.1 Issue	55
9.3.2 Execution	55
9.3.3 Write Result	56

9.4	Altri dettagli	56
9.4.1	Common Data Bus	56
9.4.2	Confronto Tomasulo vs Scoreboard	56
9.5	Esempio di funzionamento	57
9.6	Variante speculativa	59
9.6.1	Tomasulo con ROB	59
9.6.2	Le fasi di Tomasulo speculativo	59
9.6.3	Contenuto del ROB	59
9.6.4	Altre variazioni	59
IV	Oltre ILP	61
10	TLP e DLP	61
10.1	Multithreading	61
10.1.1	I thread	61
10.1.2	Cosa si introduce	61
10.2	Tipologie di TLP	61
10.2.1	Multithreading a granularità grossolana	61
10.2.2	Mutlithreading a granularità fine	62
10.2.3	Simultaneous multithreading	62
11	Multiprocessori	62
11.1	Topologie di connessione	62
11.1.1	Single bus vs network	62
11.1.2	Varie topologie di connessione	63
11.2	Confronto delle topologie di network-connected multiprocessors	64
11.2.1	Singolo Bus	64
11.2.2	Ring	64
11.2.3	Crossbar network	64
11.2.4	2D Mesh	65
11.2.5	Ipercubo	65
11.3	Modello di indirizzamento della memoria	65
11.3.1	Organizzazione logica	65
11.3.2	Organizzazione fisica della memoria	66
11.3.3	Esempi di architetture	66
11.4	Coerenza cache	67
11.4.1	Snooping	67
11.4.2	Directory based	69
11.4.3	Directory FSA	70
11.4.4	Modelli di comunicazione	71
V	Valutazione delle prestazioni	71
12	Confrontare diverse esecuzioni	71
12.1	Formule	72
12.1.1	IPC, CPI	72
12.1.2	Speedup	72
12.1.3	CPU time	72
12.1.4	MIPS	72
12.1.5	Quali metriche utilizzare	72
12.2	Legge di Amdahl	72
12.3	Basi di confronto	72
12.3.1	Microbenchmarks	72
12.3.2	Nucleo dell'applicazione	73
12.3.3	Applicazione intera	73
12.3.4	A carico effettivo	73
12.4	Prestazioni della cache	73
12.4.1	AMAT	73
12.4.2	Alcune definizioni	73

Introduzione

In questo corso vedremo alcuni dei problemi “avanzati” nelle architetture dei calcolatori:

- Come possiamo aumentare le performance riducendo il costo finale del design architetturale?
 - RISC
 - Pipeline
- Possiamo guadagnare ancora di più?
 - Branch prediction
 - Instruction level parallelism
 - Multithreading
 - Multiprocessors
- Soluzioni ancora più avanzate
 - Memory hierarchy
 - Cache organization

Riferimenti

- "*Computer Architecture, A Quantitative Approach*", John Hennessy, David Patterson, Morgan Kaufmann, Fourth Edition

Parte I

Pipeline

1 Caratteristiche della architettura MIPS

- MIPS è una architettura **RISC** (Reduced Instruction Set Computer), è basata quindi sul concetto di eseguire solo semplici istruzioni.
- Utilizza una architettura **LOAD/STORE**
 - Tutti gli operandi delle operazioni vengono spostati in registri *general purpose* della CPU e **non** possono arrivare direttamente dalla memoria.
 - Quindi, si utilizza la **LOAD** per spostare dati dalla memoria ai registri e la **STORE** per spostare dati dai registri alla memoria
- MIPS è basato su una **architettura a pipeline** che è una tecnica di ottimizzazione che si basa sulla sovrapposizione dell'esecuzione di istruzioni multiple derivate da un flusso di esecuzione sequenziale
- MIPS utilizza 32 registri

1.1 Instruction Set Architecture (base) del processore MIPS

- Istruzioni di **ALU**:

```
add $s1, $s2, $s3      # $s1 ← $s2 + $s3
addi $s1, $s2, 4        # $s1 ← $s2 + 4
```

- Istruzioni **load/store** (di *word*) da Memoria (attenzione all'ordine degli operandi):

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

- Istruzioni di **branch condizionali** (controllo di flusso):

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

- Istruzioni di **branch incondizionati**:

```
j L1                  # jump to L1
jr $s1                 # jump to address contained in $s1
```

1.2 Formato delle istruzioni MIPS a 32 bit

- **Tipo R** (Register):
 - Istruzioni ALU
- **Tipo I** (Immediate):
 - Istruzioni immediate
 - Istruzioni load/store
 - Branch condizionali
- **Tipo J** (Jump):
 - Istruzioni di jump incondizionato

1.2.1 Formato R per istruzioni ALU Register-Register

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

Opcode Identifica il tipo di istruzione ALU

rs Primo operando **sorgente**

rt Secondo operando **sorgente**

rd Registro di destinazione

shamt Quantità di shift

funct Identifica diversi tipi di istruzioni ALU

1.2.2 Formato I per istruzioni LOAD/STORE

Opcode	rs	rt	offset
6 bit	5 bit	5 bit	16 bit

Opcode Identifica il tipo di istruzione ALU

rs Registro base

rt Registro destinazione o sorgente per i dati caricati o salvati in memoria

offset Viene utilizzato (estendendolo in segno) sommato con l'indirizzo del base register

1.2.3 Formato I per istruzioni immediate ALU

Opcode	rs	rt	immediate
6 bit	5 bit	5 bit	16 bit

Opcode Identifica il tipo di istruzione ALU

rs Registro sorgente

rt Registro destinazione

immediate Contiene il valore dell'operando immediato (nel range -2^{15} e $2^{15} - 1$ a causa del primo bit di segno)

1.2.4 Formato I per istruzioni di branch condizionale

Opcode	rs	rt	address
6 bit	5 bit	5 bit	16 bit

Opcode Identifica il tipo di istruzione branch

rs Primo registro da comparare

rt Secondo registro da comparare

address Indica l'offset di parole relative alla posizione del program counter (PC-relative word address)

1.2.5 Formato J per i Jump incondizionati

Opcode	address
6 bit	26 bit

Opcode Identifica il tipo di istruzione Jump

address Contiene i 26 bit più significativi dei 32 dell'indirizzo assoluto della parola di memoria

1.3 Esecuzione delle istruzioni MIPS

Ogni istruzione del subset MIPS può essere implementata in al massimo 5 cicli di clock come segue:

1. Instruction Fetch Cycle (IF):

- (a) Invia il contenuto del *program counter* register alla *instruction memory* e legge l'istruzione corrente.
- (b) Aggiorna il *program counter* aggiungendo 4 all'indirizzo corrente (ogni istruzione sono 4 Bytes = 32 bits)

2. Instruction Decode and Register Read Cycle (ID)

- (a) Decodifica l'istruzione corrente e legge i vari *registri* specificati nell'istruzione. La lettura avviene dal register file.
- (b) L'estensione del segno nel campo offset dell'istruzione viene fatta in questo punto

3. Execution Cycle (EX)

- (a) La ALU opera sugli operandi preparati nei cicli precedenti a seconda del tipo di istruzione:
 - **Register-Register ALU Instructions:** L'ALU esegue le operazioni specificate dagli operandi
 - **Register-Immediate ALU Instructions:** L'ALU esegue l'operazione sul primo operando
 - **Memory Reference:** L'ALU aggiunge il registro base e l'offset per calcolare l'indirizzo effettivo
 - **Conditional branches:** Vengono comparati i due registri specificati e computati possibili indirizzi di salto (Branch Target Address) aggiungendo l'estensione del segno al PC.

4. Memory Access (ME):

- (a) Le istruzioni di LOAD richiedono un *accesso in lettura alla Data Memory* utilizzando l'indirizzo effettivo
- (b) Le istruzioni di STORE richiedono un *accesso in scrittura alla Data Memory* utilizzando l'indirizzo effettivo per scrivere i dati dall'indirizzo sorgente
- (c) I branch condizionali possono *aggiornare il contenuto del Program Counter* con i branch target address, se il risultato è *true*

Questa fase di memory access non è presente in tutte le istruzioni! Alcune non hanno bisogno di questa fase.

5. Write Back Cycle (WB):

- (a) Le istruzioni di LOAD scrivono i dati letti dalla memoria nel registro di destinazione
- (b) Le istruzioni ALU scrivono i risultati della ALU nel registro di destinazione

1.4 Dettagli tecnici delle istruzioni MIPS

1.4.1 Latenza delle istruzioni

Tipo di istruzione	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

1.4.2 Operazioni MIPS riassunte fase per fase

ALU instructions:

- **IF:** Fetch e incremento PC
- **ID:** Lettura dei registri sorgente ed eventuale estensione di segno dell'offset
- **EX:** Operazione ALU sui dati
- **ME:**
- **WB:** Scrittura del risultato nel register file

LOAD instructions:

- **IF:** Fetch e incremento PC
- **ID:** Lettura del registro base
- **EX:** Operazione di ALU per somma degli indirizzi
- **ME:** Operazione di lettura da memoria del valore
- **WB:** Scrittura del valore letto nel register file

STORE instructions:

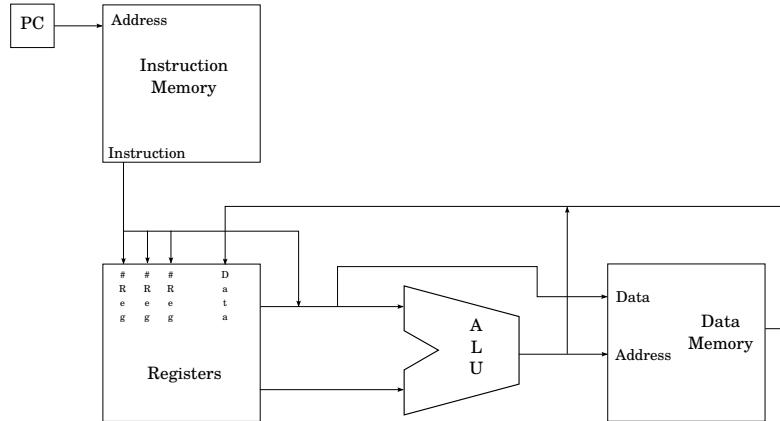
- **IF:** Fetch e incremento PC
- **ID:** Lettura del registro base e valore sorgente
- **EX:** Operazione di ALU per somma degli indirizzi
- **ME:** Scrittura in memoria del valore
- **WB:**

Conditional branch instructions:

- **IF:** Fetch e incremento PC
- **ID:** Lettura dei due registri da confrontare
- **EX:** Operazione ALU di confronto e somma del program counter in caso di branch
- **ME:** Scrittura del PC (non è nel register file)
- **WB:**

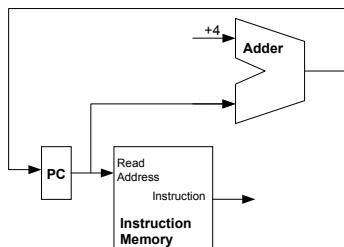
1.4.3 Schema delle implementazioni

Implementazione base del data path MIPS

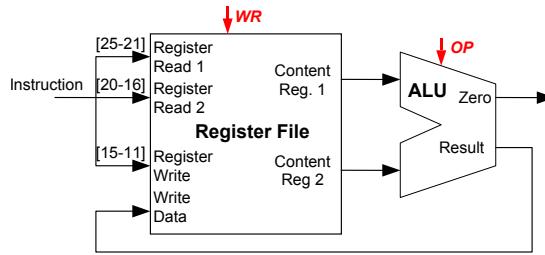


- La **instruction memory** (read-only memory) è separata dalla **data memory**
- I 32 registri *general purpose* sono organizzati in un **register file (RF)** con due porte in lettura e una porta in scrittura

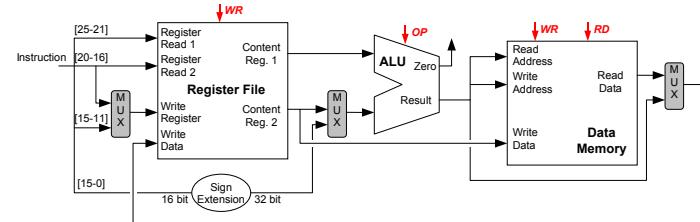
Implementazione della fase di Instruction Fetch



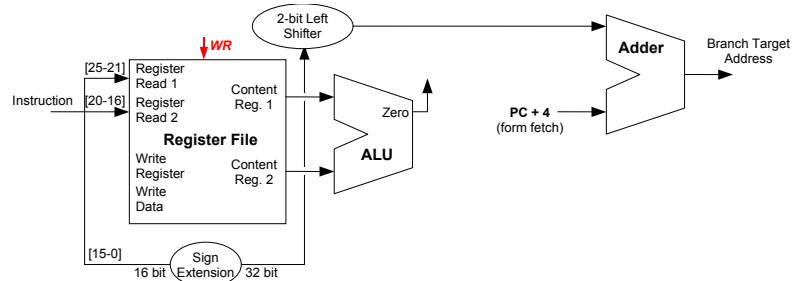
Implementazione delle istruzioni ALU



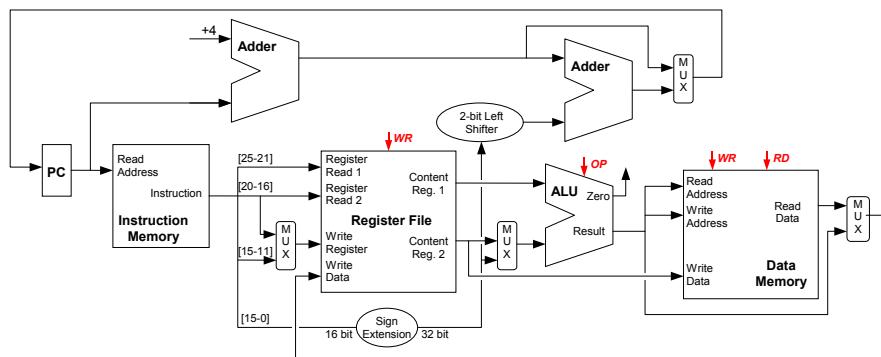
Implementazione delle istruzioni LOAD/STORE



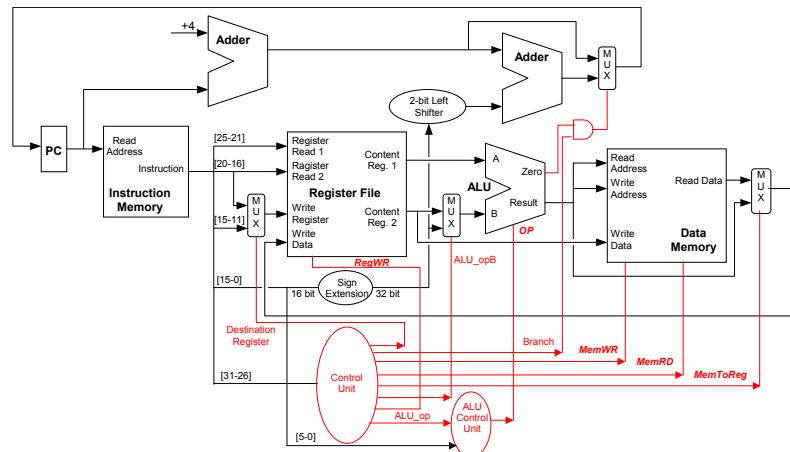
Implementazione dei branch condizionali



Implementazione del data path MIPS



Implementazione del data path MIPS con unità di controllo



1.4.4 Implementazione multi-cycle

- Abbiamo detto che l'esecuzione di una istruzione viene distribuita su più cicli (5 per MIPS)
- Il ciclo base è lungo 5 ns , quindi la durata massima di una istruzione è 10 ns
- Implementazione di una CPU multi-cycle:
 - Ogni fase della esecuzione di una istruzione richiede un ciclo di clock
 - Ogni modulo può essere utilizzato più di una volta per ogni istruzione in diversi cicli di clock. **Quindi** è possibile **condividere dei moduli**.
 - Abbiamo bisogno di registri interni per salvare i valori da utilizzare nei cicli di clock seguenti.

1.5 Pipelining

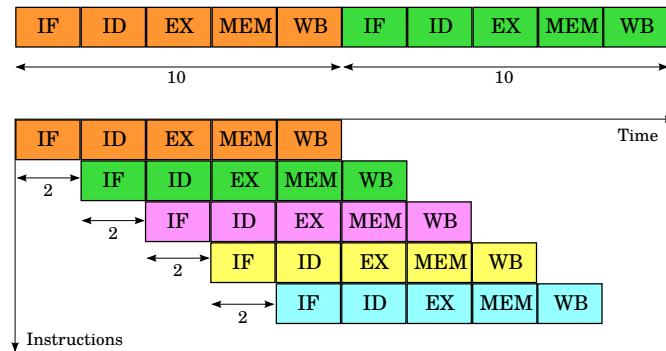
È una tecnica per l'ottimizzazione delle performance, basata sulla sovrapposizione dell'esecuzione di istruzioni multiple derivanti da un flusso di esecuzione sequenziale.

Il pipelining utilizza il parallelismo tra le istruzioni in uno stream sequenziale.

- Idea base:** L'esecuzione di una istruzione viene divisa in più fasi (pipeline stages) che richiedono una frazione del tempo necessario per completare l'istruzione.
- Gli stages sono collegati tra di loro dalla pipeline: l'istruzione entra nella pipeline da una parte, passa per vari stages e esce dall'altra parte, come su una catena d'assemblaggio.
- Vantaggio:** È una tecnica di “parallelismo” trasparente al programmatore.
- Come su una catena d'assemblaggio di automobili, in un ciclo entra una automobile e ne esce una nuova, non modifichiamo il tempo necessario per completare una automobile, ma aumentiamo il numero di automobili prodotte simultaneamente e la frequenza per completarle.

1.5.1 Esecuzione sequenziale vs. pipelining

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

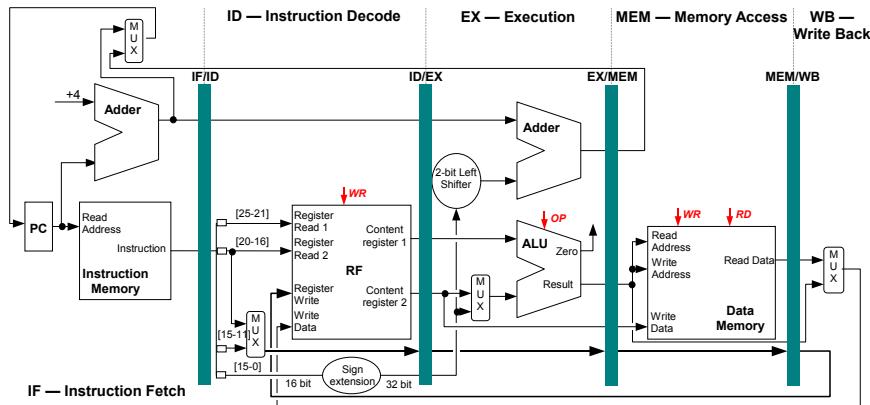


- I vari stage della pipeline devono essere **sincronizzati**: la durata di un ciclo di clock è definita dal tempo necessario allo stage più lento della pipeline.
- Se gli stages sono perfettamente bilanciati, lo **speedup ideale** che otteniamo ad utilizzare la pipeline invece dell'esecuzione sequenziale è pari al numero di stage da attraversare (in questo caso 5, quindi x5).
- Se consideriamo una CPU senza pipelining, con ciclo di clock di 8 ns e la CPU2 con 5 stages da 2ns:
 - La **latenza** di ogni istruzione peggiora (da 8 a 10 ns)
 - Però il **throughput** (numero di istruzioni completate nell'unità di tempo) aumenta di 4 volte
- Se consideriamo una CPU senza pipelining, con 5 cicli di clock di 2 ns e la CPU2 con 5 stages da 2ns:
 - La **latenza** di ogni istruzione resta invariata
 - Il **throughput** (numero di istruzioni completate nell'unità di tempo) aumenta di 5 volte

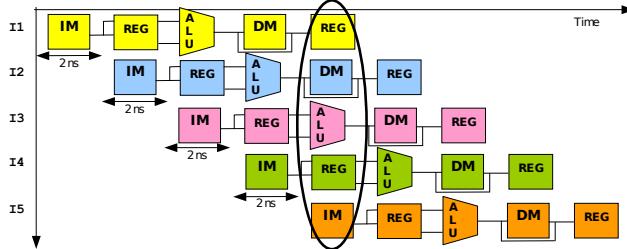
1.5.2 Le varie fasi della pipeline riviste per i vari tipi di istruzione

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

1.5.3 Implementazione della pipeline MIPS



Schema di utilizzo delle risorse nella pipeline



- IM = Instruction Memory
- REG = Register File
- DM = Data Memory

1.6 Hazards (pericoli dovuti a dipendenze)

1.6.1 Il problema

- Un *hazard* viene creato quando c'è una dipendenza tra istruzioni, le istruzioni sono vicine e quindi la pipeline cambia l'ordine di accesso agli operandi coinvolti nella dipendenza (compromettendo il risultato)
- L'*hazard* impedisce alla prossima istruzione della pipeline di essere eseguita nel suo ciclo di clock previsto
- Gli *hazards* riducono le **performance** dello speedup ideale guadagnato dalla pipeline.

Tipi di hazards

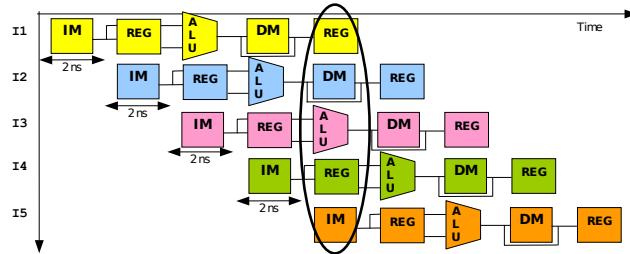
- **Structural Hazards:** Tentativo di utilizzare la stessa risorsa da diverse istruzioni simultaneamente
 - Ad esempio: memoria singola per istruzioni e dati

- **Data Hazards:** Tentativo di utilizzare un risultato prima che sia pronto
 - Ad esempio: istruzione dipendente dal risultato di una precedente istruzione ancora in pipeline
- **Control Hazards:** Tentativo di fare una decisione riguardo alla prossima istruzione da eseguire, prima che la condizione sia valutata

1.6.2 Structural Hazards

Non ce ne sono nella architettura MIPS:

- La *instruction memory* è separata dalla *data memory*.
- Il register file è utilizzato nello stesso ciclo di clock: Accesso in lettura da parte di una istruzione e accesso in scrittura da parte di un'altra istruzione.



1.6.3 Data Hazards: introduzione

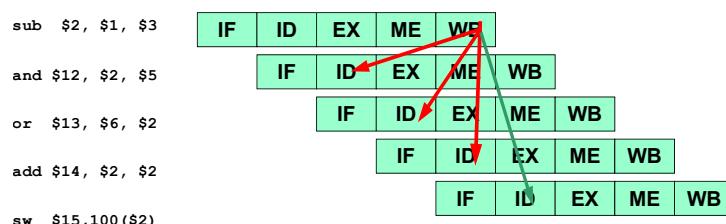
Se le istruzioni eseguite nella pipeline sono **dipendenti**, ci possono essere data hazards nel caso in cui siano troppo vicine fra loro.

Esempio:

```

sub $2, $1, $3      # Reg. $2 scritto da sub
and $12, $2, $5     # 1° operando ($2) dipende da un sub
or $13, $6, $2       # 2° operando ($2) dipende da un sub
add $14, $2, $2      # 1° ($2) & 2° ($2) dipende da un sub
sw $15, 100($2)     # Registro base ($2) dipende da un sub
  
```

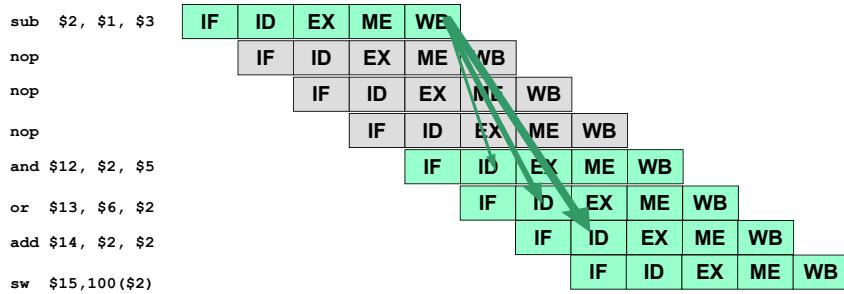
La situazione risulta abbastanza chiara dalla figura seguente. Le prime tre dipendenze costituiscono un problema in quanto richiedono che sia pronto un dato che la pipeline sta ancora lavorando.



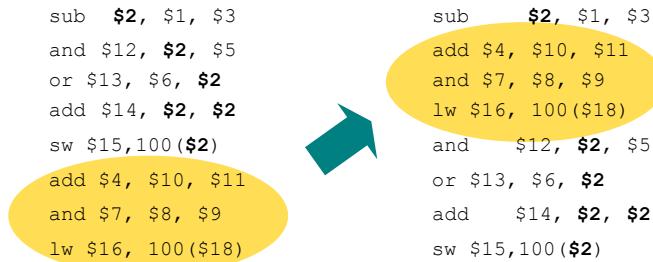
1.6.4 Data Hazards: Possibili soluzioni

- **Compilation Techniques** (in fase di compilazione):
 - Inserire delle istruzioni **NOP** (**no operation**)
 - Fare scheduling delle istruzioni (scambiare l'ordine etc...) per evitare che istruzioni correlate siano troppo vicine
 - * Il compilatore tenta di inserire istruzioni indipendenti tra quelle correlate
 - * Quando il compilatore non trova istruzioni indipendenti, inserisce **NOP**
- **Hardware Techniques:**
 - Inserimento di “bubbles” o *stalls* (stalli) nella pipeline.
 - Data *forwarding* o *bypassing*.

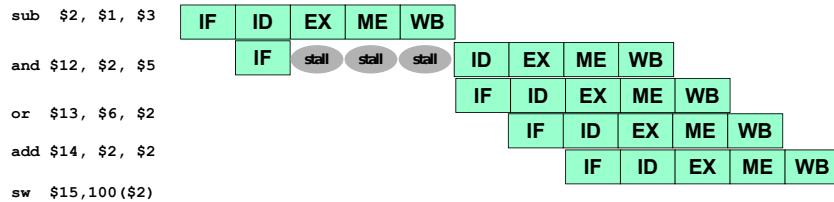
Inserimento di NOPs



Scheduling



Inserimento di stalls

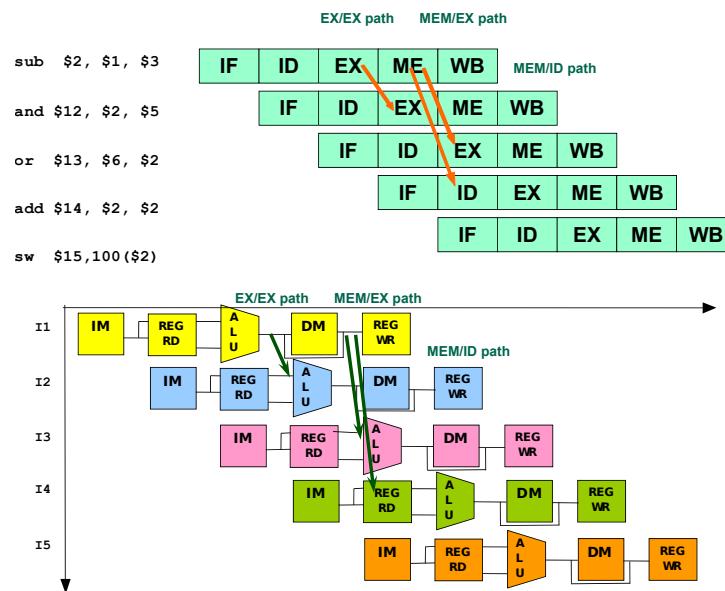


1.6.5 Forwarding

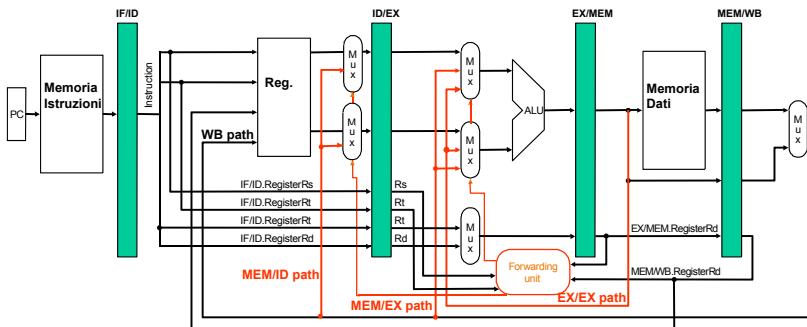
Il data forwarding utilizza risultati temporanei memorizzati nei registri della pipeline invece di aspettare il write back dei risultati nel register file.

Quindi con il forwarding abbiamo la possibilità di passare informazioni da uno stadio della pipeline ad un altro **direttamente**, senza aspettare che il dato venga riscritto (e riletto) tramite il register file (esterno alla pipeline).

Dobbiamo aggiungere multiplexers agli inputs dell'ALU per prendere input dai registri della pipeline per evitare l'inserimento di stalli.

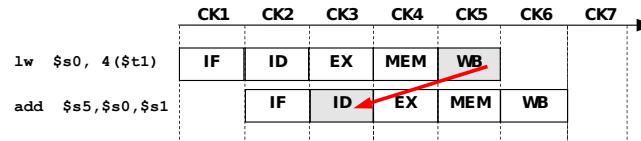


Implementazione di MIPS con la forwarding unit

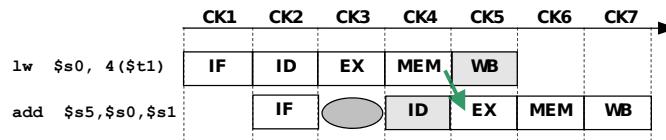


Esempi di Data Hazards: il caso Load/Use Un tipico esempio di Data Hazard è il *load/use hazard*: rientrano in questa categoria tutti gli schedule in cui l'utilizzo di un registro avviene poco dopo la sua inizializzazione tramite load. Le immagini che seguono mostrano esempi di questa situazione.

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

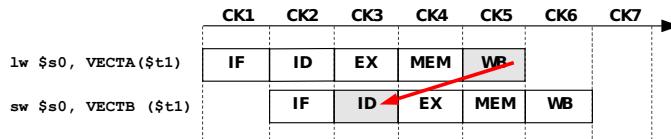


Facendo forwarding utilizzando il percorso MEM / EX è necessario 1 stall

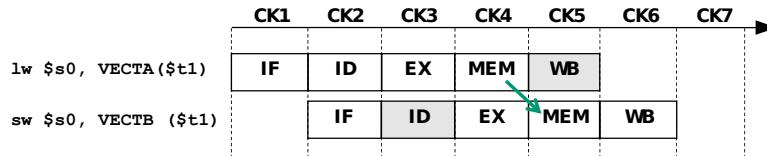


Esempi di Data Hazards: il caso Load/Store

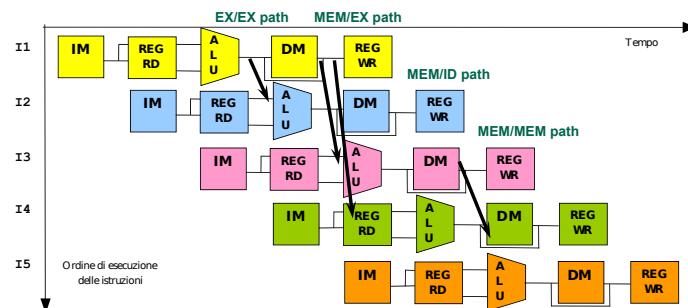
```
L1: lw $s0, VECTA($t1)      # $s0 <- M [VECTA + $t1]
L2: sw $s0, VECTB($t1)      # M [VECTB + $t1] <- $s0
```



Con un forwarding su MEM / MEM path si risolve:



I percorsi di forwarding

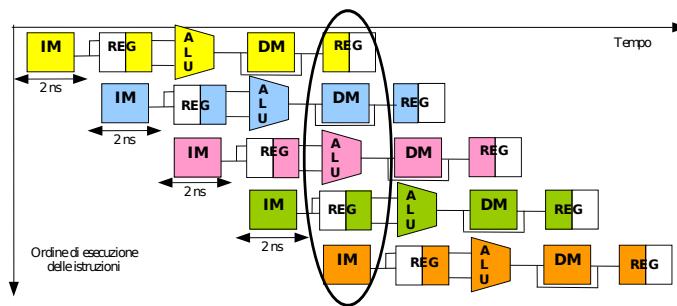


I percorsi di forwarding sono:

- EX / EX
- MEM / EX
- MEM / ID
- MEM / MEM (per LOAD/STORE)

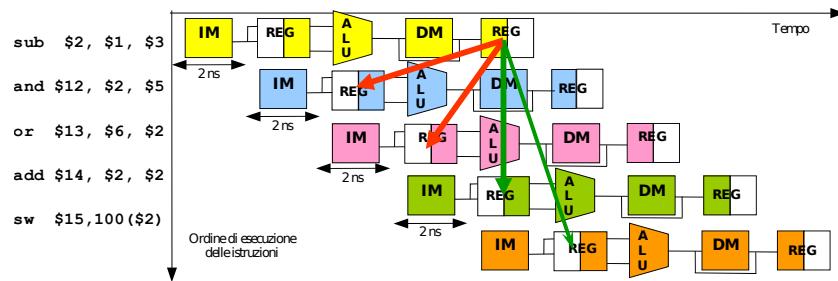
1.6.6 Ottimizzazione della pipeline

- Il register file viene utilizzato in due stages: Accesso in read durante fase ID e accesso in write durante la fase WB
- Cosa succede se read e write si riferiscono allo stesso register nello **stesso** ciclo di clock? È necessario **inserire una stall**.
- Pipeline ottimizzata: assumiamo che le read del register file accadono nella seconda metà del ciclo di clock e che le write del register file accadono nella prima metà del ciclo di clock.
 - Potremmo aver bisogno di un clock a frequenza doppia di quello normale della pipeline
 - Evitiamo così di dover inserire degli stalli.

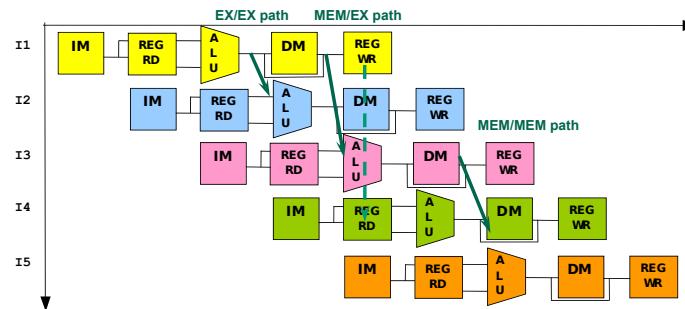


- IM = Instruction Memory
- REG = Register File
- DM = Data Memory

1.6.7 Data Hazards nella pipeline ottimizzata



In questo caso è necessario inserire **due** stalli, oppure usare i forwarding paths



- I forwarding paths sono:
 - EX / EX path
 - MEM / EX path
 - MEM / MEM path (for LOAD/STOREs)

1.6.8 Classificazione dei Data Hazards (RAW - WAW - WAR)

RAW - Read After Write Istruzione $n + k$ tenta di leggere un registro sorgente scritto prima che l'istruzione n lo abbia scritto nel register file.

- Esempio:

```
add $r1, $r2, $r3
sub $r4, $r1, $r5
```

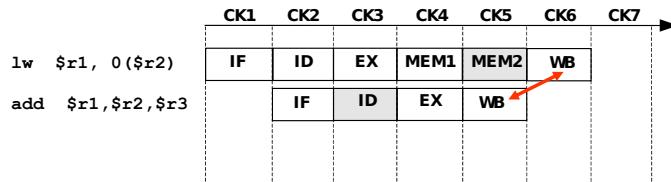
Utilizzando il forwarding è sempre possibile risolvere questo conflitto senza introdurre stalls, eccetto per i load/use hazards

WAW - Write After Write L'istruzione $n + k$ tenta di scrivere un operando di destinazione prima che sia stato scritto dall'istruzione precedente n .

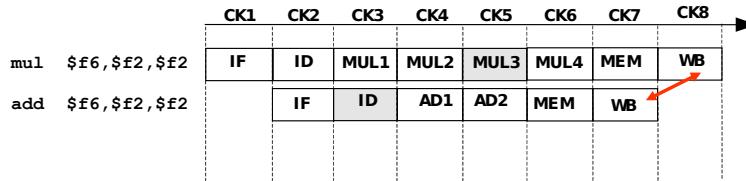
Quindi le operazioni di write vengono eseguite nell'ordine sbagliato

- Questo tipo di hazard può presentarsi sulla pipeline MIPS quando si estendono operazioni multi-ciclo per eseguire o accedere alla memoria dati.

Ad esempio, se assumiamo che la scrittura su registro delle istruzioni ALU avvenga nel quarto stage e che le istruzioni di LOAD richiedano 2 stages (MEM1 e MEM2) per accedere alla memoria dati, possiamo avere:



- Se assumiamo che le operazioni ALU floating point richiedano una esecuzione multi-ciclo, abbiamo:



WAR - Write After Read L'istruzione $n+k$ tenta di scrivere un operando di destinazione prima che venga letto dall'istruzione precedente n .

Quindi l'istruzione n legge il valore sbagliato.

- Esempio:

```
sw $y, 0($x)      # sw deve leggere $x
addi $x, $x, 4    # addi scrive $x
```

Gli hazards di tipo WAR non possono accadere nella pipeline MIPS perché le operazioni di read avvengono sempre nello stage ID, mentre le operazioni di write avvengono sempre nello stage WB

- Come prima, se assumiamo che la scrittura di registri delle istruzioni ALU avviene nel quarto stage e che abbiamo bisogno due stages per accedere alla memoria dati, alcune operazioni possono leggere gli operandi troppo tardi nella pipeline.

1.7 Problemi di performance nella Pipeline

Come abbiamo visto la Pipeline aumenta il throughput ma non la latenza delle istruzioni.

In realtà il pipelining aumenta leggermente la latenza di tutte le istruzioni a causa di overhead e sbilanciamento tra le latenze delle istruzioni.

1.7.1 Metriche base di Performance

$IC = \text{Instruction Count}$

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

$$CPI = \text{Clock Per Instruction} = \frac{\#Clock Cycles}{IC} = \frac{(IC + \#Stall Cycles + 4)}{IC}$$

$$MIPS = \frac{f_{clock}}{(CPI \cdot 10^6)}$$

MIPS: Million Instructions Per Second, è una unità di misura delle performance

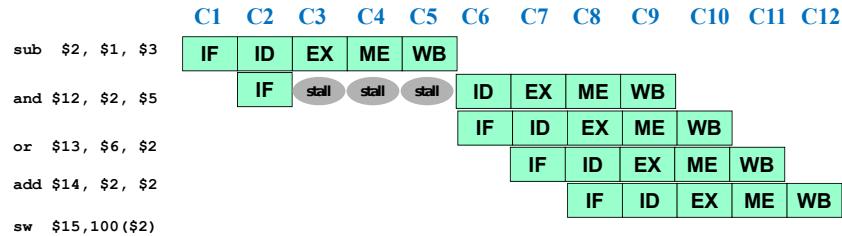
Esempio

$$IC = \text{Instruction Count} = 5$$

$$\#Clock Cycles = 12$$

$$CPI = 2.4$$

$$MIPS = 208.3$$



1.7.2 Altre metriche di performance

$$IC_{\text{per iterazione}} = m$$

$$\#Clock Cycles_{\text{per iterazione}} = IC_{\text{per iterazione}} + \#Stall Cycles_{\text{per iterazione}} + 4$$

$$CPI_{\text{per iterazione}} = \frac{(IC_{\text{per iterazione}} + \#Stall Cycles_{\text{per iterazione}} + 4)}{IC_{\text{per iterazione}}} = \frac{(m + k + 4)}{m}$$

$$MIPS_{\text{per iterazione}} = \frac{f_{clock}}{CPI_{\text{per iterazione}} \cdot 10^6}$$

1.7.3 Metriche di performance asintotiche

Consideriamo n iterazioni di un ciclo composto da m istruzioni per iterazione che richiedono k stalli per iterazione (nel seguito AS starà per “asintotico”):

$$IC_{AS} = \text{Instruction Count}_{AS} = m \cdot n$$

$$\#Clock Cycles = IC_{AS} + \#Stall Cycles_{AS} + 4$$

$$CPI_{AS} = \lim_{n \rightarrow \infty} \left(\frac{IC_{AS} + \#Stall Cycles_{AS} + 4}{IC_{AS}} \right) = \frac{IC_{AS} + \#Stall Cycles_{AS}}{IC_{AS}} = \frac{(m + k)}{m}$$

$$MIPS_{AS} = \frac{f_{clock}}{CPI_{AS} \cdot 10^6}$$

Il CPI ideale per un processore con pipeline sarebbe 1, ma gli stalli fanno degradare le performance, quindi abbiamo:

$$CPI_{\text{medio PIPE}} = CPI_{\text{Ideal}} + Pipe Stall Cycles_{\text{per istruzione}} = 1 + Pipe Stall Cycles_{\text{per istruzione}}$$

I cicli di stallo per iterazione sono dovuti a structural hazards, data hazards, control hazards e memory stalls

1.7.4 Speedup della pipeline

$$\text{Pipeline speedup} = \frac{\text{Avarage execution time}_{\text{senza pipeline}}}{\text{Avarage execution time}_{\text{con pipeline}}}$$

$$\text{Pipeline speedup} = \frac{\text{Avarage CPI}_{\text{senza pipeline}}}{\text{Avarage CPI}_{\text{con pipeline}}} \cdot \frac{\text{Clock Cycle}_{\text{senza pipeline}}}{\text{Clock Cycle}_{\text{con pipeline}}}$$

Se ignoriamo l'overhead sul tempo di clock a causa del pipelining e assumiamo che i vari stage sono perfettamente bilanciati, il tempo di clock di due processori può essere uguale, quindi:

$$\text{Pipeline speedup} = \frac{\text{Avarage CPI}_{\text{senza pipeline}}}{1 + \text{Pipe Stalls per instruction}}$$

- **Caso semplice:** Tutte le istruzioni prendono lo stesso numero di cicli, che deve essere uguale al numero di stadi della pipeline (chiamata *profondità della pipeline*):

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline Stalls per instruction}}$$

Se non ci sono stalli nella pipeline (caso ideale), questo ci porta al risultato intuitivo che il pipelining può aumentare le performance man mano che aumenta la profondità della pipeline.

- Qual'è l'impatto di performance dei **branch condizionali**?

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipe Stall Cycles per instruction due to branches}} = \frac{\text{Pipeline depth}}{1 + \text{Branch Frequency} \cdot \text{Branch Penalty}}$$

2 Tecniche di predizione dei branch

2.1 Istruzioni di branch condizionali

Il branch viene preso soltanto se una certa **condizione** viene **soddisfatta**.

L'**indirizzo di destinazione** del *branch* viene salvato nel *program counter* (PC) al posto dell'indirizzo dell'istruzione successiva (nello stream sequenziale di istruzioni).

- Esempi di branch per il processore MIPS: **bne** e **beq**

```
beq $s1, $s2, L1          # Vai a L1 se $s1 == $s2  
bne $s1, $s2, L1          # Vai a L1 se $s1 != $s2
```

Abbiamo già visto com'è strutturata l'istruzione di branch in memoria (è una istruzione di tipo I). Le fasi per la sua esecuzione sono:

- Fetch delle istruzioni e modifica di PC
- Lettura dei registri dal register file
- ALU operation: per confrontare i registri e derivare il **Branch Outcome** (branch “taken” o “not taken”).
- Computazione del **branch target address** ($PC + 4 + offset$) (ricordiamoci che offset viene esteso in segno).
- Il valore di **PC** viene scritto.

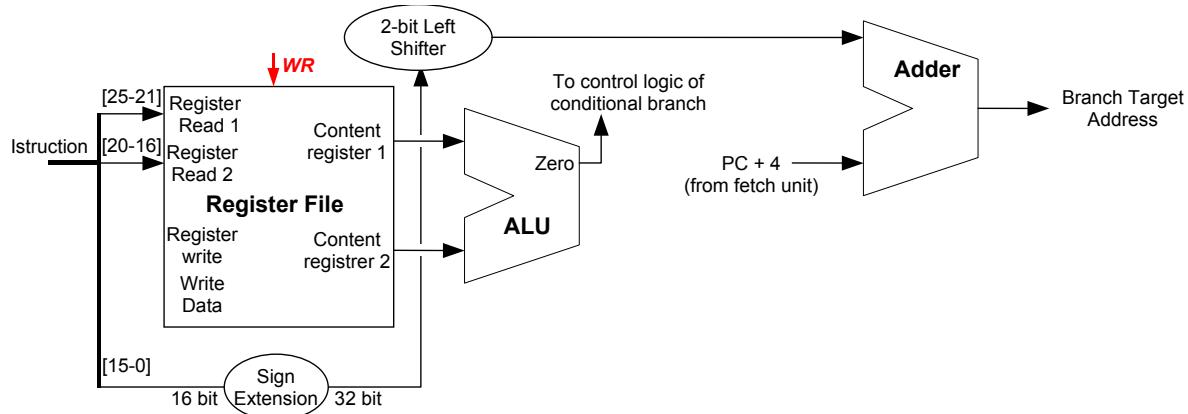
Abbiamo già visto che in MIPS ogni istruzione si divide in 5 stages:

1. IF - Instruction Fetch
2. ID - Instruction Decode
3. EX - Execution
4. ME - Memory Access
5. WB - Write Back

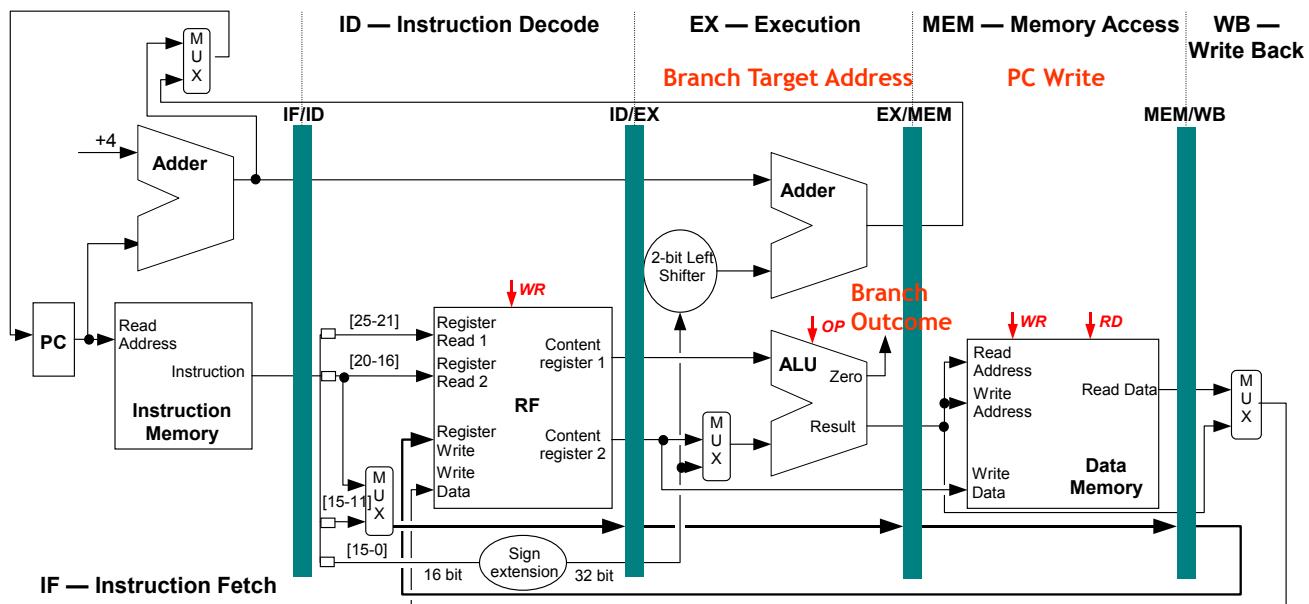
Nel caso della nostra istruzione di branch, il **branch outcome** e il **branch target address** saranno pronti alla fine dello stage EX (stage 3).

I branch condizionali vengono completati quando il PC è stato riscritto, nello stage 4.

- Le risorse del processore per eseguire i branch condizionali:



- Implementazione della pipeline a 5 stage MIPS:



2.2 Il problema dei control hazards

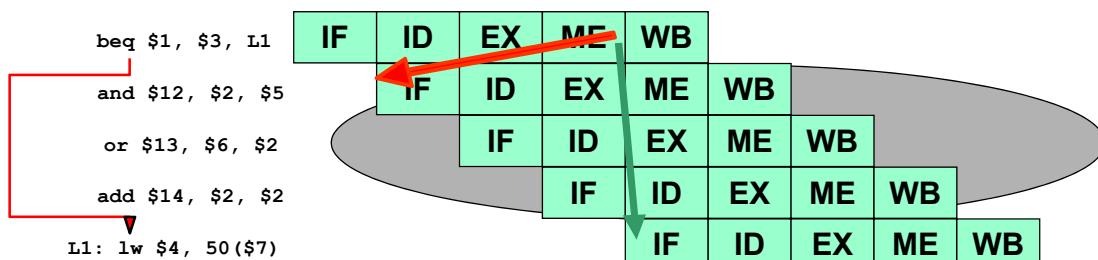
Un **control hazard** è un tentativo di prendere una decisione sulla prossima istruzione da leggere prima che la condizione del branch venga valutata.

- I **control hazards** nascono quando branch condizionali e altre istruzioni che **modificano il valore di PC** vengono messe all'interno della **pipeline**
- I **control hazards** riducono le performance dallo speedup ideale guadagnato dal pipelining perché potrebbero rendere necessari degli stalli nella pipeline.

Per far funzionare la pipeline dobbiamo prendere una nuova istruzione ad ogni ciclo di clock, ma la decisione di branch (cambiare o non cambiare PC per la **prossima** istruzione) viene presa durante lo stadio MEM (stadio 4).

Questo **ritardo** per determinare la prossima istruzione è chiamato appunto **control hazard** o **conditional branch hazard**. Se il branch decide di cambiare il valore di PC si dice **branch taken**, altrimenti è un **untaken**.

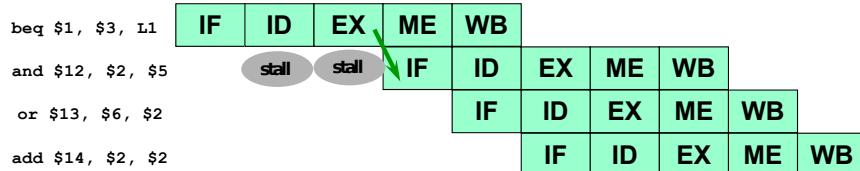
Esempio



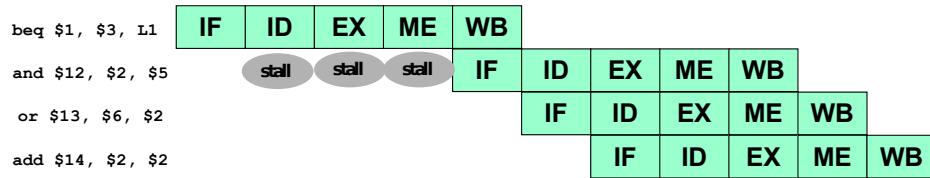
- Nel momento in cui il branch decide se deve cambiare o non cambiare PC, ci sono già le 3 istruzioni seguenti caricate nella pipeline.
- Se il branch è **not taken** non c'è nessun problema. Se però è taken ho già eseguito parte delle 3 istruzioni seguenti! Quindi è necessario fare un “**flush**” delle successive 3 istruzioni

2.3 Soluzioni al problema del control hazard

- Assunzione **conservativa**: Metto in stallo la pipeline fino a quando la decisione non è stata presa (si chiama anche **stalling until resolution**) e poi prendo il flusso corretto di istruzioni
 - Se utilizziamo il forwarding: lo faccio in 3 cicli di clock



- Senza forwarding: in 2 cicli di clock



- Se però il branch era **untaken** abbiamo sprecato 3 preziosi cicli di clock che ci causano una grossa riduzione del throughput.
- Possiamo assumere che il branch sia **untaken** e fare flush delle successive 3 istruzioni solo se pensiamo che sia **taken**.

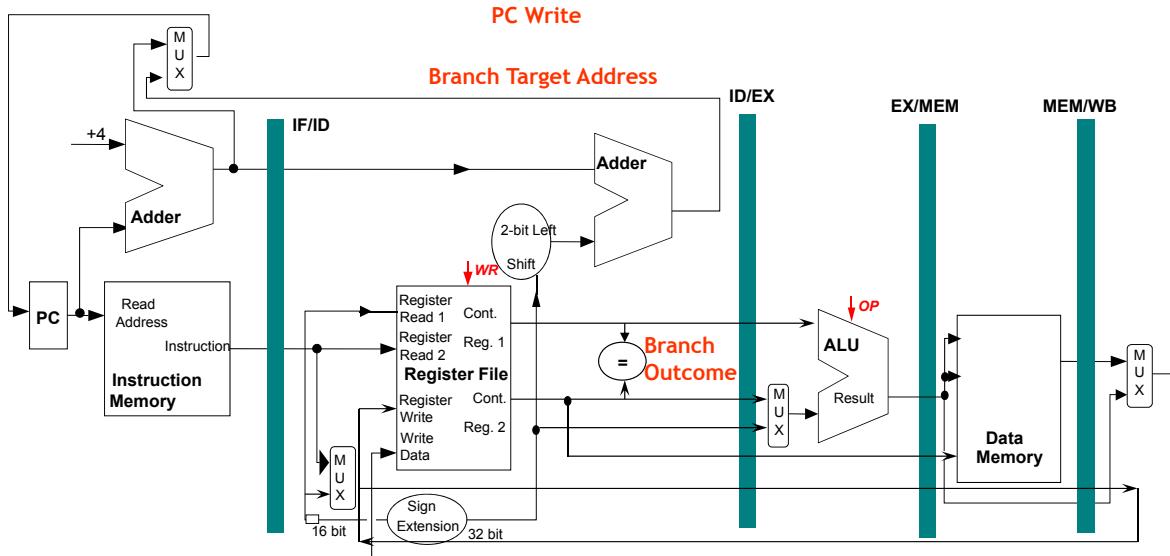
2.3.1 Early evaluation del PC

Per aumentare le performance nel caso di hazards dovuti ai branch, dobbiamo aggiungere nuove risorse hardware per:

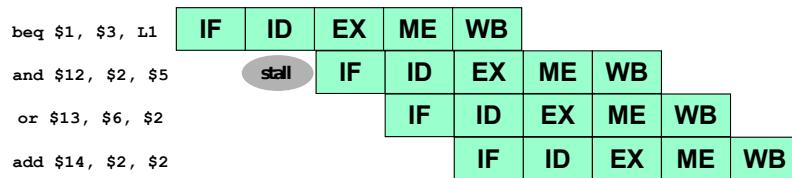
- Confrontare registri per derivare il **branch outcome** (risultato del branch)
- Computare il **branch target address**
- Aggiornare il registro PC

Tutte queste 3 operazioni devono essere fatte appena possibile nella pipeline.

I processori MIPS comparano registri, computano indirizzi di branch e aggiornano il PC durante lo **stadio ID**.



- **Assunzione conservativa:** Ogni branch costa uno stall per fetchare il corretto flusso di istruzioni



- Quali sono le conseguenze di fare questa “*early evaluation*” ?

- Nel caso della istruzione di **add** (o **load**) seguita da una **branch** che testa il risultato della add: dobbiamo introdurre uno stall prima dello stadio ID del branch per permettere il forwarding del risultato dallo stadio EX della istruzione precedente.

- **Esempio:**



- Con la decisione di branch fatta durante lo stadio ID c'è una **riduzione dei costi** associati ad ogni branch (**branch penalty**)
 - * Abbiamo bisogno solo un ciclo di clock di stallo dopo ogni branch
 - * Oppure il flush di **una** sola istruzione che segue il branch
- Il ritardo di un ciclo per ogni branch comporta comunque una perdita dal 10% al 30% a seconda della frequenza del branch

$$\text{Pipeline stall cycles per instruction}_a \text{ causa dei branch} = \text{Branch frequency} \cdot \text{Branch penalty}$$

- Esamineremo alcune tecniche per affrontare questa perdita di performance

2.4 Tecniche di predizione dei branch

In generale il problema dei branch diventa più importante per processori con pipeline profonda perché il costo delle predizioni scorrette aumenta (i branch vengono risolti diversi stage dopo lo stadio ID).

- **Le performance** delle tecniche di predizione di branch dipendono da:
 - L'**accuratezza** misurata in termini di performance di predizioni scorrette date dal predittore
 - Il **costo** di una predizione scorretta misurata in termini di tempo perso per eseguire **istruzioni inutili**
- Dobbiamo anche considerare la **frequenza dei branch** all'interno del flusso di codice da eseguire. Ovviamente se ci sono poche branch per istruzione avremo poca perdita di performance.

Ci sono fondamentalmente **due metodi di predizione** dei branch:

- **Tecniche di predizione statica:** A compile time decidiamo quale politica attuare in corrispondenza di ciascun branch
 - Questo tipo di predizione viene utilizzata nei processori dove si riesce a prevedere l'esito dei branch a compile time
- **Tecniche di predizione dinamica:** A runtime prendiamo decisioni sulle predizioni dei branch, che possono cambiare durante l'esecuzione del programma.

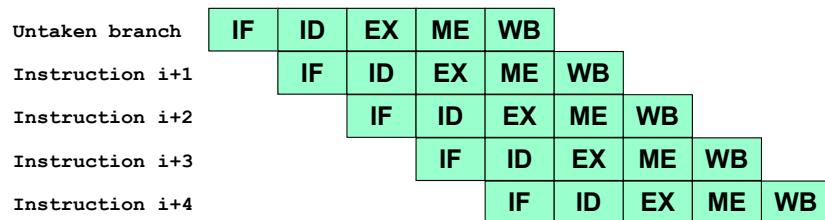
In entrambi i casi è necessario fare molta attenzione e tenersi pronti ad attuare una procedura di recovery quando l'esito effettivo del branch rivela una misprediction (predizione errata).

2.4.1 Tecniche di predizione statica

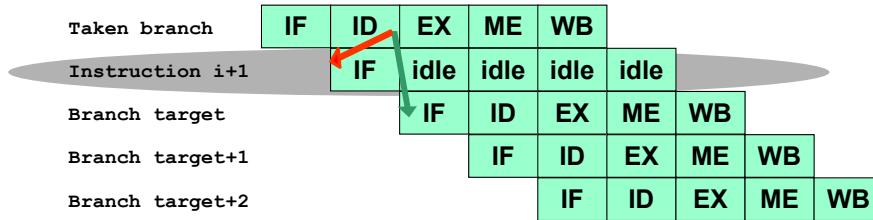
1. Branch **always not taken (Predicted Not Taken)**
2. Branch **always taken (Predicted Taken)**
3. **Backward taken forward not taken (BTFNT)**
4. **Profile-Driven Prediction**
5. **Delayed Branch**

Branch always (not) taken Assumiamo che il branch non sia mai taken, quindi lasciamo continuare il flusso di istruzioni come se la condizione di branch non fosse soddisfatta.

- Se la predizione è **corretta** preserviamo le performance:



- Se la predizione è **errata**: dobbiamo fare flush della prossima istruzione già fetchata (trasformiamo l'istruzione letta in precedenza in una NOP) e riavviamo l'esecuzione prendendo l'istruzione al *branch target address*. Abbiamo quindi una **penalità di un ciclo**. Si ricorda che l'architettura MIPS implementa l'ottimizzazione Early Evaluation del PC.



Uno schema alternativo è quello di considerare ogni branch come **taken**.

Non appena il branch target address viene computato, assumiamo il branch come **taken** e iniziamo a fare fetch e esecuzione del branch target.

Note sulla Branch Always Taken

- Questo tipo di predizione ha senso per pipelines dove il branch target address è noto prima di conoscere il risultato del branch.
- Nella pipeline MIPS non conosciamo il *branch target address* prima di conoscere il **branch outcome**. Quindi per MIPS questa tecnica non è utile.

Backward Taken Forward Not Taken (BTFNT) La predizione viene fatta sulla “direzione” dei branch (cioè se l’indirizzo destinazione a cui puntano è prima o dopo l’indirizzo dell’istruzione attuale):

- **backward-going branches**: considerati *taken*
- **forward-going branches**: considerati *not taken*

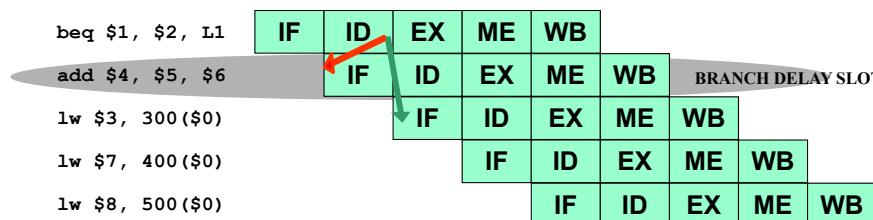
L’idea alla base è che quando abbiamo dei branch che puntano *forward* stiamo eseguendo degli if e quindi stiamo valutando eccezioni nel codice che si presume vengano effettivamente eseguite meno di frequente; in presenza di branch che puntano *backward* significa che siamo all’interno di un loop¹, e quindi (nella maggior parte dei casi), il branch sarà taken per proseguire il loop.

Profile-driven prediction La tecnica profile-driven consiste nel raccogliere statistiche sul numero di volte che ciascun branch viene preso e non preso. A valle di questo profiling svolto su un sufficiente numero di simulazioni (o esecuzioni) del software ogni branch viene marcato con il valore di predizione più probabile.

Questo metodo può anche integrarsi con “consigli” del compilatore.

Delayed Branch Technique Il compilatore riordina il codice e inserisce staticamente negli slot di “delay” dei branch istruzioni indipendenti dal branch stesso, le quali vengono eseguite sia che il branch risulti taken sia che risulti untaken.

- Se assumiamo che il branch delay sia 1 ciclo (come nel caso di MIPS ottimizzato), abbiamo solo 1 slot di delay.
 - In realtà è possibile avere processori con pipeline che utilizzano degli slot di delay più lunghi.
 - Però quasi tutti i processori che utilizzano la tecnica di delayed branch hanno un singolo slot di delay. Ciò avviene perché è particolarmente difficile per il compilatore trovare più istruzioni da inserire in più di uno slot.
- Il compilatore MIPS schedula sempre una istruzione indipendente dopo una istruzione di branch.
- Esempio: una istruzione di add viene schedulata dopo una istruzione beq, per essere eseguita nel *branch delay slot*.



- Ci sono **4 modi** in cui il branch delay slot può essere schedulato dal compilatore:

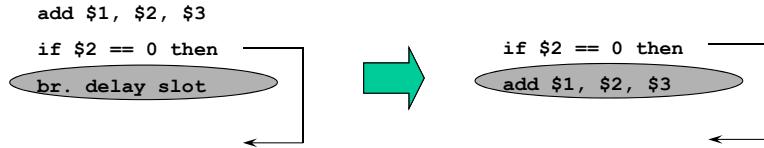
1. From before
2. From target
3. From fall-through
4. From after

¹si ricorda che nella maggior parte dei casi anche i cicli con controllo in testa (while, for) vengono tradotti con cicli a controllo in coda

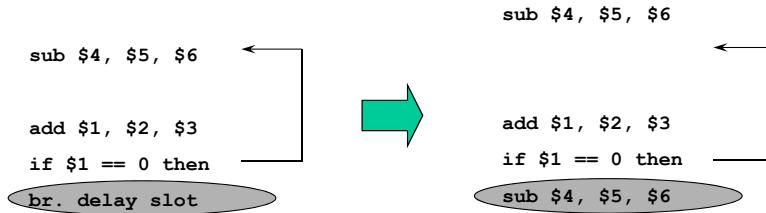
Delayed branch technique: from before Il branch delay slot viene schedulato con una istruzione indipendente che era prima dell'istruzione bi branch.

L'istruzione nel branch delay slot viene sempre eseguita (perché in ogni caso la dobbiamo eseguire non essendo una istruzione che è dopo).

- Esempio:

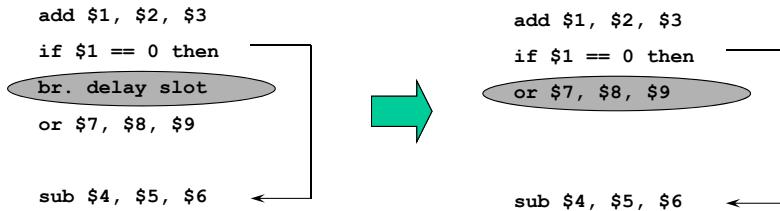


Delayed branch technique: from target



In questo caso non possiamo spostare l'istruzione di **add** dopo l'**if** perché il risultato (\$1) viene utilizzato per prendere decisioni sul branch. Però in questo caso possiamo mettere nel branch delay slot l'istruzione da eseguire in caso di branch taken se il branch target address è backward (come nel caso dei loop).

Delayed branch technique: from fall-through



È la tecnica duale del “*from target*”

Anche in questo caso non possiamo spostare l'istruzione di **add** perché il branch utilizza \$1 che è già presente nella add.

In questa strategia iniziamo ad eseguire il comando dopo il branch (quindi come se fosse a priori **not taken**). Questo è utile quando abbiamo alta probabilità di branch non presi.

- Per rendere l'ottimizzazione legale nei casi “*from target*” e “*from fall-through*” deve essere consentito eseguire l'istruzione spostata quando il branch va nella direzione opposta a quella prevista. Per “consentito” intendiamo che l'istruzione inserita nel branch delay slot può essere eseguita ma il “lavoro” viene soltanto sprecato e il programma può continuare correttamente anche dopo aver eseguito quella istruzione inutile, ovvero l'istruzione nel delay slot non ha sporcato lo stato dell'esecuzione. Per esempio va bene se il registro destinazione dell'operazione inutile è un registro utilizzato solo nel ramo errato del flusso di esecuzione.
 - In generale, i compilatori sono in grado di riempire circa il 50% dei delayed branch slots con istruzioni valide e utili. Gli slot rimanenti sono riempiti con operazioni **NOP**.
 - Nelle pipelines più profonde il delayed branch è più lungo di un ciclo, quindi è necessario trovare altre istruzioni ed è ancora più difficile.
- Le principali **limitazioni** del delayed branch scheduling nascono da:
 - Restrizioni sulle istruzioni che possiamo inserire nel branch delay slot.
 - L'abilità del compilatore di predire staticamente il risultato dei branch.
- Per migliorare le predizioni del compilatore molti processori hanno introdotto la **cancelling** o “*nullifying branch*”. In questo tipo speciale di branch l'istruzione include al suo interno anche un bit che indica la direzione in cui è previsto che il branch vada. Così facendo abbiamo che:
 - Quando il branch agisce come previsto: l'istruzione nel branch delay slot è eseguita normalmente.

- Quando il branch non agisce come previsto: l'istruzione nel branch delay slot viene trasformata in una NOP.
- L'architettura MIPS ha l'istruzione **branch-likely** che si comporta come una branch *cancel-if-not-taken*.
 - L'istruzione nel branch delay slot viene eseguita se il branch è taken.
 - L'istruzione nel branch delay slot non viene eseguita (viene trasformata in NOP) se il branch è untaken
 - È un approccio utile per backward branches (come i branch utilizzati per nei loop)

2.4.2 Tecniche di predizione dinamica dei branch

Idea base: utilizzare i comportamenti dei branch precedenti per prevedere il futuro.

Utilizziamo hardware per prevedere dinamicamente il risultato di un branch. La predizione dipenderà sul comportamento del branch a runtime e cambierà se il branch cambia il suo comportamento durante l'esecuzione.

Partiamo con uno schema di predizione di branch semplice e poi esamineremo approcci che aumentano l'accuratezza.

I meccanismi principali di predizione sono due:

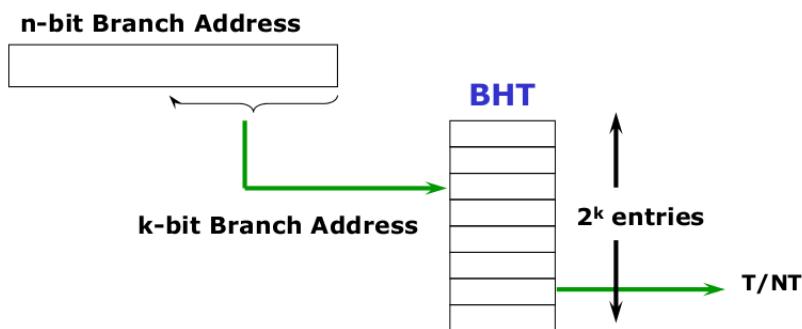
- **Branch outcome predictor:** per prevedere la direzione di un branch (taken o not taken)
- **Branch target predictor (BTP) o branch target buffer :** per prevedere il branch target address in caso di branch taken

Questi moduli sono utilizzati dall'unità di *instruction fetch* per prevedere la prossima istruzione da leggere nella cache delle istruzioni (I-cache)

- Se il branch è not taken → PC viene incrementato
- Se il branch è taken → BTP (il predittore) dà l'indirizzo di destinazione

Branch History Table - BHT (o Branch Prediction Buffer) È una tabella che contiene 1 bit per ogni entry che dice se il branch è stato **taken** recentemente o untaken. Questa tabella è indicizzata, come indice si usa la parte meno significativa del campo indirizzo dell'istruzione di branch.

- Predizione: abbiamo un indizio che si assume essere corretto e iniziamo a fare il fetch nella direzione prevista.
 - Se scopriamo che la predizione era sbagliata, il bit di predizione viene invertito e ri-salvato. La pipeline viene ripulita e la sequenza corretta viene eseguita.
- La tabella non ha tags (ogni accesso è un "HIT") e il bit di predizione può essere messo da un'altra branch con lo stesso indice, ma non importa. La predizione **fornisce solo una traccia**.



Accuratezza della Branch History Table Abbiamo una predizione errata quando:

- la predizione è sbagliata per quel branch OPPURE
- se lo stesso indice è stato utilizzato per due branch diversi.

Per risolvere il secondo problema è sufficiente aumentare il numero di righe nella tabella o utilizzare una buona funzione di hash.

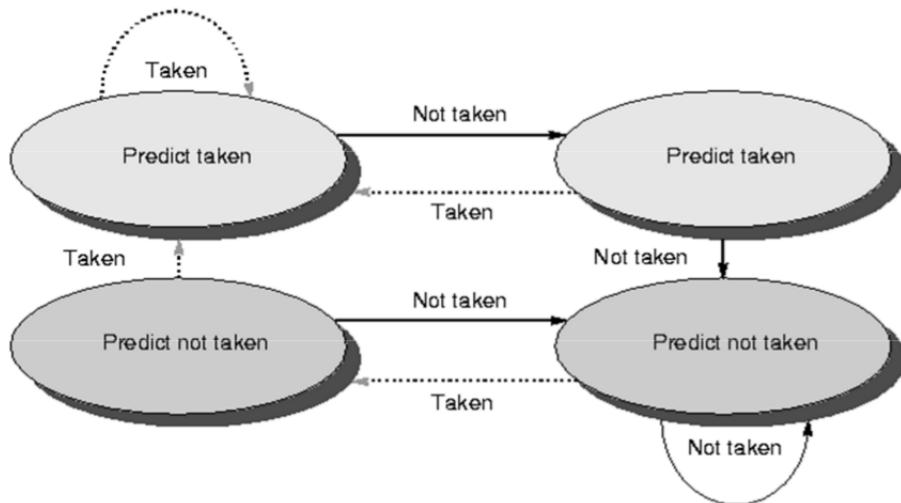
Altri problemi si hanno

- In una branch che fa parte di un loop, poiché utilizziamo un solo bit per la tabella, abbiamo una predizione errata due volte quando il loop termina (invece di una volta sola).
- Nell'ultimo ciclo del loop, poiché la predizione dice "taken" mentre noi dobbiamo uscire dal loop.
- Quando ri-entriamo di nuovo nello stesso loop, sbagliamo di nuovo a prevedere il branch perché avevamo cambiato il bit quando eravamo usciti.

Sembra un problema trascurabile, ma se consideriamo che, per esempio, abbiamo 2 predizioni scorrette e 8 corrette significa che l'accuratezza della predizione è solo dell'80%!

Modifica utilizzando 2 bit

- Se utilizziamo due bit invece di uno possiamo fare che per modificare la predizione sono necessari due "errori" invece di uno solo.
- In questo modo nel caso precedente del loop sbagliamo solo 1 volta la predizione → 90% di accuratezza.
- I due bit vengono utilizzati per memorizzare uno dei 4 stati di una macchina a stati finiti.



Branch History Table a n-bit

- I valori vanno da 0 a $2^n - 1$. Quando il contatore rappresentato dagli n bit è superiore o uguale a $2^n - 1$, il branch viene previsto come **taken**, altrimenti **untaken**
- Il contatore viene incrementato quando il branch è taken e decrementato quando il branch è untaken.
- Diversi studi su predittori a n -bit mostrano come il caso a 2-bit sia comunque sufficiente e abbastanza performante.

Correlare i predittori dei branch **Idea:** i comportamenti dei branch sono correlati, cioè c'è correlazione tra le varie branch oltre che tra la stessa branch.

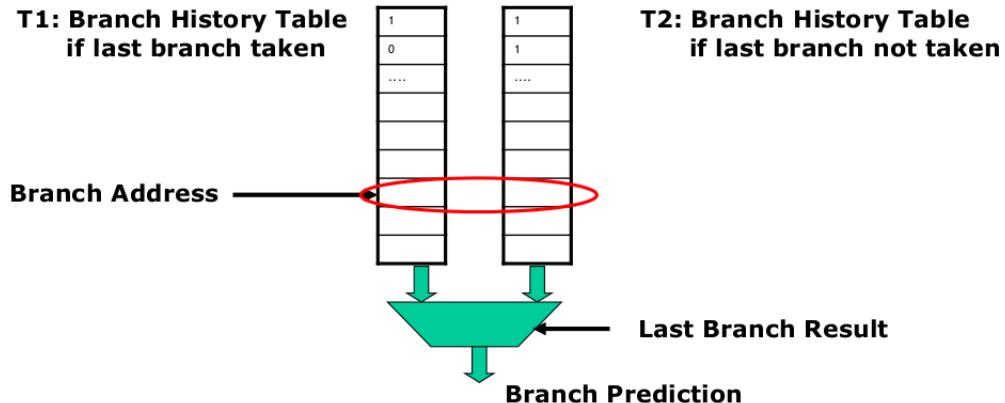
Esempio

```

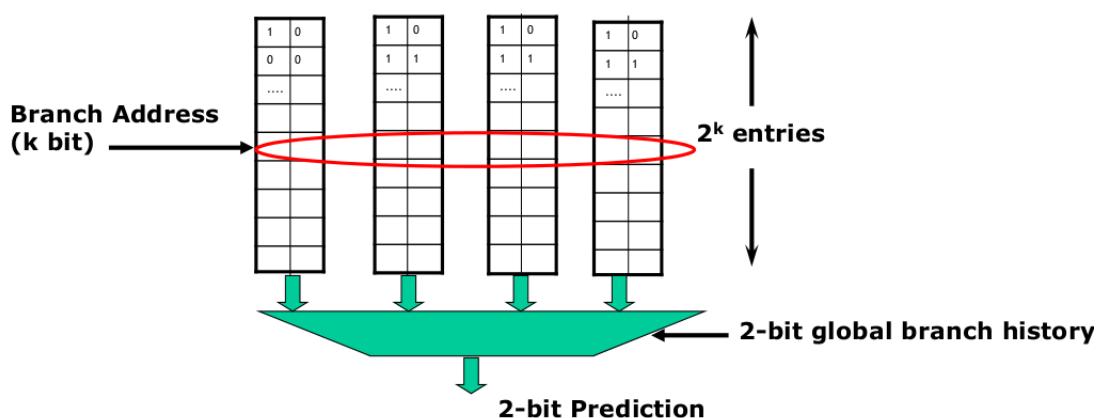
If (a==2) a = 0; bb1
L1: If (b==2) b = 0; bb2
L2: If (a!=b) {}; bb3
L1: subi r3,r1,2
      bnez r3,L1; bb1
      add r1,r0,r0
      subi r3,r2,2
      bnez r3,L2; bb2
      add r2,r0,r0
      sub r3,r1,r2
      beqz r3,L3; bb3
L2:
L3:

```

- Il branch **bb3** è correlato ai branch precedenti **bb1** e **bb2**. Se i branch precedenti sono entrambi *not taken*, allora **bb3** sarà taken ($a \neq b$).
- Questo tipo di predittori sono chiamati *correlating branch predictors* o *2-level predictors*.
- Un *correlating predictor* (1,1) è un predittore ad 1 bit con 1 bit di correlazione. Il comportamento dell'ultimo branch è utilizzato per scegliere da quale predittore osservare la predizione finale.
- Esempio:

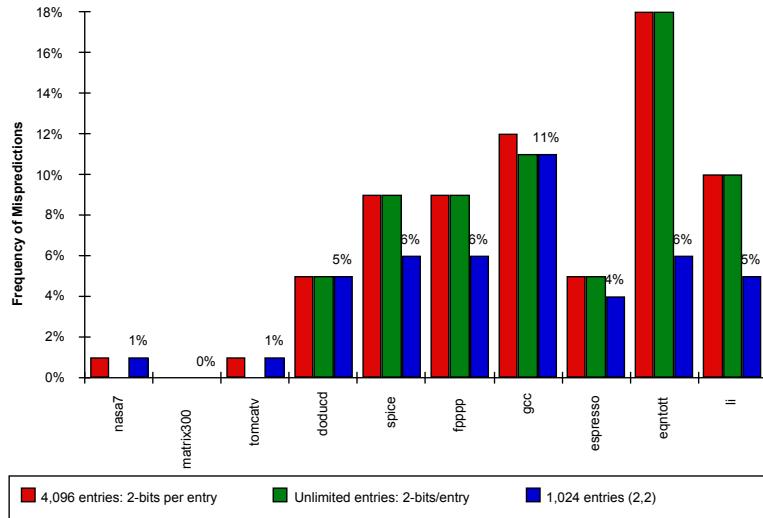


- Registro se gli ultimi k branch sono stati taken o not taken.
- Il branch viene previsto in base a quello che è successo nel branch precedente selezionando appropriatamente un bit dalla tabella:
 - * Una predizione viene utilizzata se l'ultima branch eseguita (non necessariamente la stessa branch) era taken.
 - * L'altra predizione viene utilizzata se l'ultima branch eseguita (non necessariamente la stessa branch) era not taken.
- Più in generale, un *correlating branch predictor* (m, n) registra le ultime m branch in 2^m Branch History Tables, ognuna delle quali è un predittore a n bit. Il buffer di predizione dei branch può essere indirizzato utilizzando una concatenazione di bits meno significativi del branch address insieme con m bit della history globale (quindi delle ultime m branch).
- Esempio: Correlating Branch Predictor (2,2): Abbiamo $2^2 = 4$ Branch History Tables da 2 bit ciascuna.



- 64 entries, indice di 6 bit composto da 2 bit di history globale e 4 bit per la parte meno significativa dell'indirizzo del branch.
- Ogni BHT è composta da 16 entries di 2 bit ciascuna. Il branch address di 4 bit viene utilizzato per scegliere 4 entries (una riga).
 - I 2 bit di history globale vengono utilizzati per scegliere una di 4 entries in una riga (una delle 4 BHTs).
- Un predittore a 2 bit senza global history lo possiamo vedere come un predittore (0,2).

Accuratezza dei correlating predictors

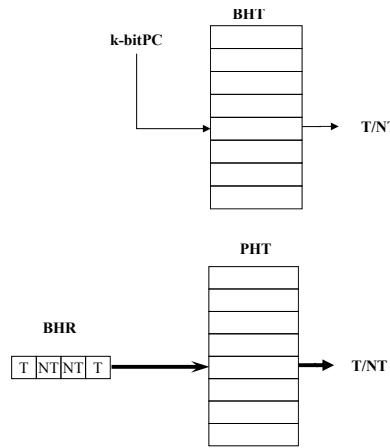


Two-level Adaptive Branch Predictors

- Il primo livello della history è registrato in uno o più registri da k bit, chiamati **Branch History Register (BHR)**, che regista i risultati delle ultime k branch.
- Il secondo livello di history è registrato in uno (o più) tabelle chiamate **Pattern History Table (PHT)**, di contatori a 2 bit
- La BHR viene utilizzata per indirizzare il PHT per selezionare quale contatore a 2 bit utilizzare.
- Una volta che il contatore a 2 bit viene selezionato, la predizione viene fatta con lo stesso metodo come nello schema del contatore a 2 bit.

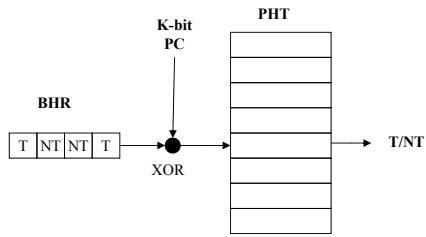
Global Adaptive Predictor

- BHT:** Predittore locale
 - Indirizzato dai bit meno significativi del PC (branch address)
- GAs:** Predittore locale e globale
 - Predittore a 2 livelli: PHT indirizzata dal contenuto di BHR (history globale)



- GShare:** Local XOR Global information

È indirizzato dallo **XOR** tra i bit meno significativi del PC (**branch address**) e il contenuto di **BHR** (history globale)



Branch Target Buffer BTB - Branch Target Buffer (Branch Target Predictor) è una cache che memorizza il target address previsto per la prossima istruzione dopo il branch.

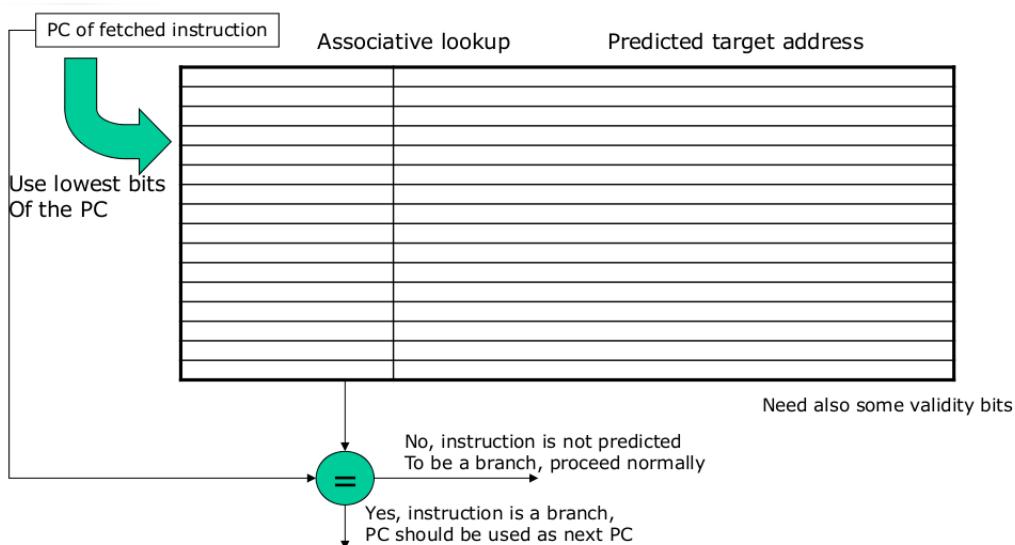
- Possiamo accedere al BTB nello stadio IF utilizzando l'indirizzo istruzione dell'istruzione fetchata (un possibile branch) come indice per questa cache.

- la tipica entry del BTB è:

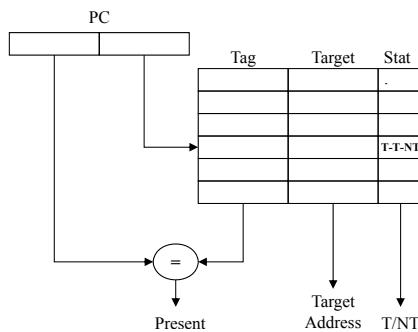
Exact Address of a Branch	Predicted Target Address
---------------------------	--------------------------

– predicted target address è espresso relativamente al PC.

- Struttura del Branch Target Buffer:



- Nella BTB dobbiamo memorizzare il target address previsto solo per i branch taken.
- La entry della BTB è: **tag + Target address previsto (PC-relative) + Bits per lo stato di predizione**



Commenti sulla branch prediction Senza la branch prediction, la quantità di parallelismo è abbastanza limitata poiché è possibile agire solo all'interno di un singolo **basic block**, cioè un pezzo di codice privo di branches.

Le tecniche di predizione di branch ci aiutano a migliorare il parallelismo. Nei metodi di predizione che abbiamo mostrato, facciamo fetch, issue ed execute di istruzioni successive come se le predizioni fossero sempre corrette, avendo però sempre pronto un metodo per risolvere le situazioni di predizione errata.

Queste speculazioni possono essere supportate dal **compilatore** o dall'**hardware**.

Parte II

ILP

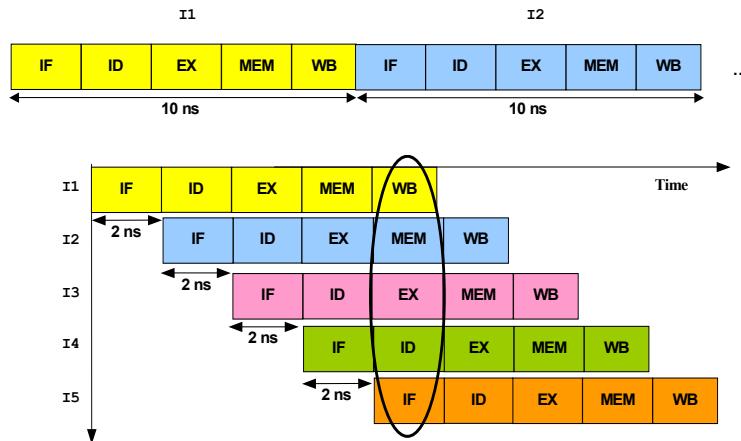
3 Instruction Level Parallelism

3.1 Portare il parallelismo a livello di istruzione

In una macchina dotata di pipeline, il CPI viene derivato come:

$$CPI_{pipeline} = CPI_{ideale} + \text{Structural Stalls} + \text{Data Hazards Stalls} + \text{Control Stalls} + \text{Memory Stalls}$$

- La riduzione di qualunque termine a destra ci permette di rendere $CPI_{pipeline}$ sempre più vicino a CPI_{ideale}
 - Aumentiamo le istruzioni per clock: $IPC = \frac{1}{CPI}$
- **Caso migliore:** il throughput massimo dovrebbe essere di una istruzione per clock: $IPC_{ideal} = 1$ e $CPI_{ideal} = 1$



3.1.1 Riassunto sulla pipeline

- Gli Hazards limitano le performance
 - **Hazards strutturali:** risolviamo con più risorse hardware
 - **Hazards dati:** risolviamo con il forwarding e compiler scheduling.
 - **Hazards controllo:** early evaluation, branch delay slot, static branch prediction, dynamic branch prediction.
- Aumentare la lunghezza della pipeline aumenta l'impatto degli hazards
- La pipeline ci aiuta ad **aumentare il throughput, non la latenza.**

3.1.2 Riassunto tipi di Data hazards

Data una sequenza di istruzioni

$$r_k \leftarrow (r_i) \ op \ (r_j)$$

Determiniamo i vari tipi di dipendenze che esistono tra di esse:

Data-dependence la seconda istruzione deve aspettare il risultato della prima per avere il valore del registro operando. Può causare un **RAW hazard**.

Anti-dependence la prima istruzione utilizza come operando un registro che viene usato come destinazione dall'istruzione successiva. La prima istruzione deve leggere il registro prima che venga sovrascritto. Può causare un **WAR hazard**.

Output-dependence due istruzioni scrivono sullo stesso registro destinazione. Deve essere mantenuto l'ordine di commit delle istruzioni. Può causare un **WAW hazard**.

Data-dependence

$$\begin{array}{l} r_3 \leftarrow (r_1) \text{ op } (r_2) \\ r_5 \leftarrow (r_3) \text{ op } (r_4) \end{array} \quad \begin{array}{l} \text{Read-after-Write} \\ (\text{RAW}) \text{ hazard} \end{array}$$

Anti-dependence

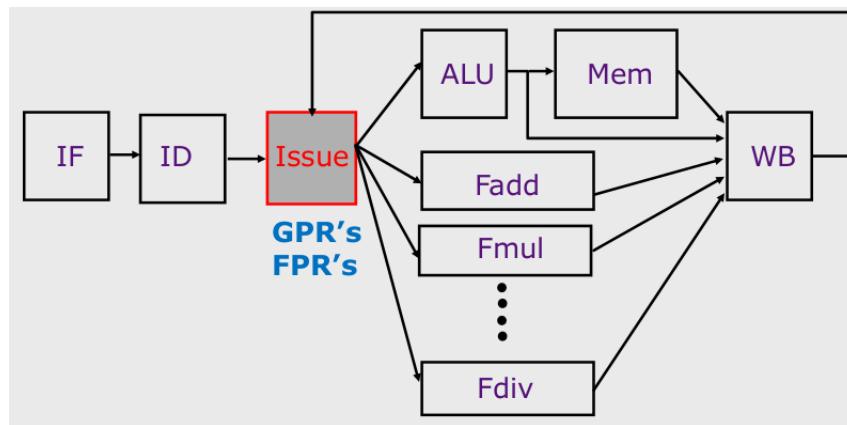
$$\begin{array}{l} r_3 \leftarrow (r_1) \text{ op } (r_2) \\ r_1 \leftarrow (r_4) \text{ op } (r_5) \end{array} \quad \begin{array}{l} \text{Write-after-Read} \\ (\text{WAR}) \text{ hazard} \end{array}$$

Output-dependence

$$\begin{array}{l} r_3 \leftarrow (r_1) \text{ op } (r_2) \\ r_3 \leftarrow (r_6) \text{ op } (r_7) \end{array} \quad \begin{array}{l} \text{Write-after-Write} \\ (\text{WAW}) \text{ hazard} \end{array}$$

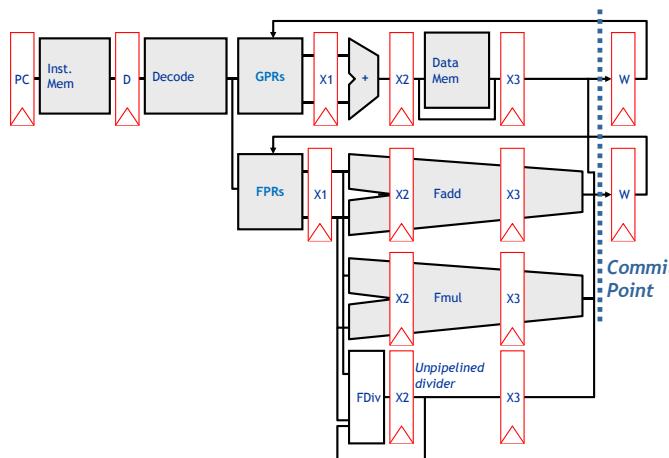
3.1.3 Evoluzioni della pipeline

Complex pipelining Aggiungendo risorse hardware per parallelizzare ulteriormente arriviamo a uno schema di questo tipo



- Unità floating point con alta latenza o “parzialmente pipelined”
- Diverse funzioni e unità di memoria
- Sistemi di memoria con tempo di accesso variabile

Complex In-order Pipeline Il precedente schema non garantisce l'*in-order commit*. Aggiungiamo quindi uno stadio allo schema



- **Delay writeback** di modo che tutte le operazioni abbiano la stessa latenza nello stage **W**
 - Le porte di write non sono mai affollate (una istanza in e una istanza out per ogni ciclo)
 - Le istruzioni vengono inviate in ordine (di modo da gestire le eccezioni in modo preciso)

3.1.4 Altri concetti base e definizioni

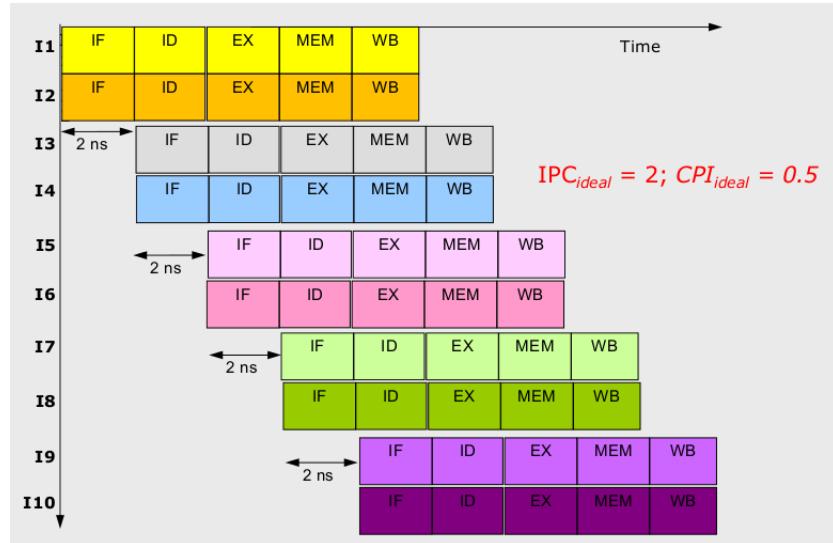
- Per raggiungere le performance più elevate (per una data tecnologia), maggior parallelismo deve essere estratto dal programma. In altre parole: **multiple-issue**
- Le dipendenze devono essere individuate e risolte e le istruzioni devono essere riordinate (**scheduled**) in modo da ottenere il parallelismo di istruzioni eseguibili più elevato possibile con le risorse a disposizione.

Instruction Level Parallelism

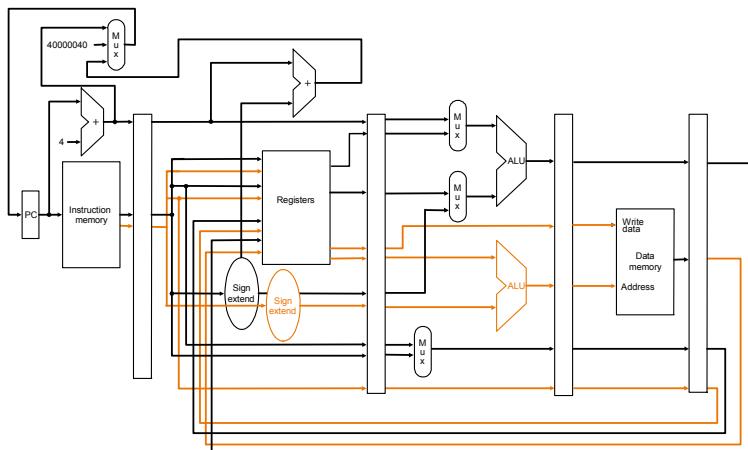
ILP Instruction Level Parallelism. *Sfruttare potenziali overlap di esecuzione tra istruzioni non correlate tra loro.*

L'overlapping è possibile solo quando:

- Non ci sono hazards strutturali
- Non ci sono hazards RAW, WAR o WAW.
- Non ci sono control hazards.



2-issue MIPS Pipeline architecture



- Due istruzioni inviate per clock:
 - 1 istruzione ALU o BR
 - 1 istruzione LOAD/STORE
- In una pipeline multiple-issue, il CPI ideale dovrebbe essere $CPI_{ideal} < 1$
- Se consideriamo per esempio un processore a 2 issue, nel caso migliore il throughput massimo sarebbe quello di completare due istruzioni per ciclo di clock:

$$IPC_{ideal} = 2$$

$$CPI_{ideale} = 0.5$$

3.2 Approfondimenti sulle dipendenze

Determinare le dipendenze tra le istruzioni è un punto critico per riuscire a definire la quantità di parallelismo esistente in un programma. Se due istruzioni sono dipendenti, non possono essere eseguite in parallelo: devono essere seguite in modo da essere solo parzialmente sovrapposte.

Ci sono tre diversi tipi di dipendenze:

- **Name dependences**
- **Data dependences (or True Data Dependences)**
- **Control dependences**

3.2.1 Name dependences

Sono quelle dipendenze che si presentano quando due istruzioni utilizzano lo stesso registro o locazione di memoria (da cui il termine “*name*”), ma non c’è un flusso di dati tra le istruzioni associate a quel *name*.

- Esistono due tipi di name dependences tra una istruzione *i* che precede una istruzione *j* nell’ordine del programma sono:
 - **Antidependence**: Quando *j* scrive un registro o locazione di memoria che l’istruzione *i* legge (può generare un **WAR**). L’ordine delle istruzioni originale deve essere preservato per assicurarsi che *i* legga il valore corretto.
 - **Output Dependence**: Quando *i* e *j* scrivono lo stesso registro o locazione di memoria (può generare un **WAW**). L’ordine originale delle istruzioni deve essere preservato per assicurarsi che il valore scritto alla fine corrisponda a quello di *j*.
- Le name dependences non sono delle vere data dependences poiché non c’è nessun valore (data flow) che viene trasmesso tra le istruzioni.
- Se il *name* può essere cambiato in base alla semantica del programma, le istruzioni non vanno in conflitto
- Le dipendenze in locazioni di memoria sono difficili da individuare (problema della “**memory disambiguation**”) perché due indirizzi possono riferirsi alla stessa locazione ma possono apparire diversi (ad esempio uno a puntamento diretto, un altro per offset). I conflitti sui registri invece sono più semplici da individuare.
- Il **renaming** può essere fatto dal compilatore dinamicamente o dall’hardware.

3.2.2 Data dependences e Hazards

Una dipendenza dati o name può generare un data hazard (RAW, WAW, WAR) ma la presenza o meno di un hazard (e del numero di stalli necessari a risolverlo) dipende strettamente dalla pipeline considerata.

- **RAW** hazards corrispondono a true data dependences.
- **WAW** hazards corrispondono a dipendenze di output.
- **WAR** hazards corrispondono a anti-dipendenze.

Le dipendenze sono una proprietà del programma, mentre gli hazards sono una proprietà della pipeline.

3.2.3 Control dependences

Le dipendenze di controllo determinano l’ordinamento delle istruzioni. L’ordinamento viene preservato con due proprietà:

- Assicurarsi che nel flusso di istruzioni, le istruzioni che si presentano prima di un branch vengano eseguite effettivamente prima del branch
- Individuare i control hazard per assicurarsi che una istruzione (che è dipendente da un branch) non sia eseguita fino a quando la direzione del branch non è nota.

Garantendo la control dependence possiamo preservare l’ordine del programma, però la **control dependence non è la proprietà critica** che deve essere preservata.

3.2.4 Program Properties

Ci sono due proprietà che sono critiche per la correttezza di un programma (e sono normalmente preservate mantenendo le dipendenze di controllo e dati):

1. **Data flow**: Il flusso corretto di dati tra le varie istruzioni, che produce il risultato corretto.
2. **Exception behavior**: Preservare il comportamento delle eccezioni significa che qualunque modifica nell’ordine di esecuzione delle istruzioni non deve modificare il modo in cui le eccezioni vengono sollevate nel programma.

3.3 Register Renaming

Si tratta di una tecnica che consente di utilizzare diversi registri fisici per la stessa locazione logica.

Quando una operazione desidera accedere in scrittura ad un registro, viene prima cambiato nome al registro in modo che essa scriva invece su un nuovo registro di appoggio. Viene tenuta traccia dello storico dei registri usati tramite una tabella di renaming; questa tabella viene interpellata ogni volta che una operazione richiede la lettura di un registro, la tabella fornisce l'identificativo del registro fisico equivalente aggiornato più di recente. La struttura della tabella di renaming varia a seconda di come il renaming è effettuato.

Questo consente di evitare conflitti di tipo WAW e WAR.

Il register renaming può avvenire:

- dal **compilatore**

- Effettuato a compile time.
- In questo caso molto diffusa è la tecnica di portare il codice assembly in forma Static Single Assignment (**SSA**).
 - * In questa forma ogni registro viene assegnato solamente una volta. Tutte le successive scritture vengono redirette verso nuovi alias.

- dall'**hardware in modo implicito**

- Effettuato a runtime.
- L'**hardware** implicitamente rinomina i registri copiando ogni volta il valore del registro in una locazione diversa.
 - * Si veda Algoritmo di Tomasulo.

- dall'**hardware in modo esplicito**

- Effettuato a runtime.
- Si introduce una traduzione “al volo” dei registri mentre vengono caricate le istruzioni.
 - * Necessita di hardware e logica supplementare per la tabella di renaming.
 - * Si veda 8.5.
- Disaccoppia il renaming dallo scheduling.

4 Evoluzione dello scheduling ILP: processori superscalari

Ci sono due strategie per supportare ILP:

- **Dynamic Scheduling**: L'hardware si occupa di identificare il parallelismo.
- **Static Scheduling**: Il software si occupa di identificare il parallelismo.

Gli approcci “hardware intensive” dominano il mercato desktop e server.

4.1 Assunzioni

- Consideriamo dei processori *single-issue*
- Lo stadio di *instruction fetch* precede lo stadio di issue e può fare fetch in un *instruction register* (o direttamente in una coda di istruzioni da eseguire).
- Le istruzioni sono poi inviate all'*issue* dall'instruction register (o dalla coda).
- Lo stadio di esecuzione può richiedere più di un ciclo, a seconda del tipo di operazione.

4.2 Dynamic Scheduling

Il problema sono le dipendenze dati che non possono essere nascoste con un bypass o forwarding a causa degli stalli della pipeline.

La **soluzione** è quella di permettere alle **istruzioni dietro uno stall di procedere**. L'hardware ri-arrangia dinamicamente l'esecuzione delle istruzioni per ridurre gli stalli.

- Viene aggiunta la possibilità della **out-of-order execution** e **out-of-order completion (commit)**.

Esempio di Dynamic Scheduling

```
DIVD F0, F2, F4  
ADDD F10, F0, F8      #RAW F0  
SUBD F12, F8, F14
```

- RAW Hazard: ADDD va in stallo per F0 (aspettando che DIVD vada in commit).
- SUBD dovrebbe andare in stallo anche se non c'è nessuna dipendenza dati o altro sulla pipeline (se non ci fosse il dynamic scheduling).

Ergo la nostra idea è quella di permettere a SUBD di procedere (*out-of-order execution*).

4.2.1 Pro e Contro

- I vantaggi:
 - Permette di gestire alcuni casi dove le dipendenze sono sconosciute a *compile time*.
 - Permette di semplificare i compilatori.
 - Permette al codice compilato di girare efficientemente su una pipeline diversa (portabilità).
- Gli svantaggi (costi):
 - Aumenta parecchio la complessità dell'hardware.
 - Aumenta il consumo energetico del processore.
 - Abbiamo il rischio di generare eccezioni (interrupt) "imprecise".

4.2.2 Riassumendo

Nella pipeline semplice abbiamo hazards a causa delle dipendenze dati che non possono essere risolti con forwarding nella pipeline. Con il dynamic scheduling invece l'hardware riordina l'esecuzione delle istruzioni di modo da ridurre gli stalli, mantenere il flusso dati e il comportamento delle eccezioni. L'esempio tipico è il **processore superscalare**.

- Le istruzioni vengono fetcate ed inviate nell'ordine del programma (*in-order issue*).
- L'esecuzione inizia appena gli operandi sono disponibili, è possibile avere *out-of-order execution*.
- L'esecuzione *out-of-order* introduce la possibilità di data (name) hazards di tipo **WAR** o **WAW**.
- L'esecuzione *out-of-order* implica che anche il completamento sarà *out-of-order*, a meno di un buffer che riordina i risultati prima che vengano scritti.

4.3 Static Scheduling

I compilatori utilizzano algoritmi sofisticati per lo scheduling del codice per ottenere *instruction level parallelism*.

Un **basic block** (una sequenza di codice senza branch, eccetto per l'inizio e la fine) è solitamente **piccolo** e la quantità disponibile di parallelismo all'interno un basic block è comunque poca.

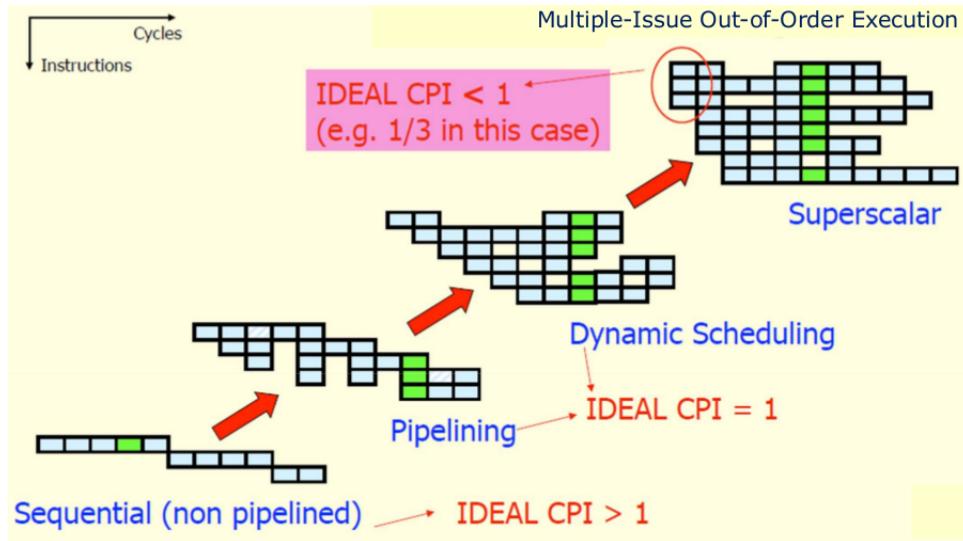
- Ad esempio per un tipico programma MIPS la frequenza media di branch è da 15% e 25%, che significa che ci sono tra 4 e 7 istruzioni da eseguire tra due branch.
- Le **dipendenze dati** possono limitare ancora di più la quantità di parallelismo che possiamo ottenere da un basic block.
- Quindi, per ottenere dei miglioramenti di performance sostanziali, dobbiamo sfruttare il **parallelismo tra diversi basic blocks**.

L'esempio tipico sono i processori **VLIW** (**Very Long Instruction Word**) che si aspettano un codice privo di dipendenze dal compilatore.

Abbiamo però alcuni **limiti** di utilizzo dello scheduling statico:

- I branch non sono prevedibili.
- La latenza della memoria è variabile (non possiamo prevedere i cache miss).
- Può aumentare la dimensione del segmento codice.
- Aumenta la complessità del compilatore.

4.3.1 Evoluzione del parallelismo



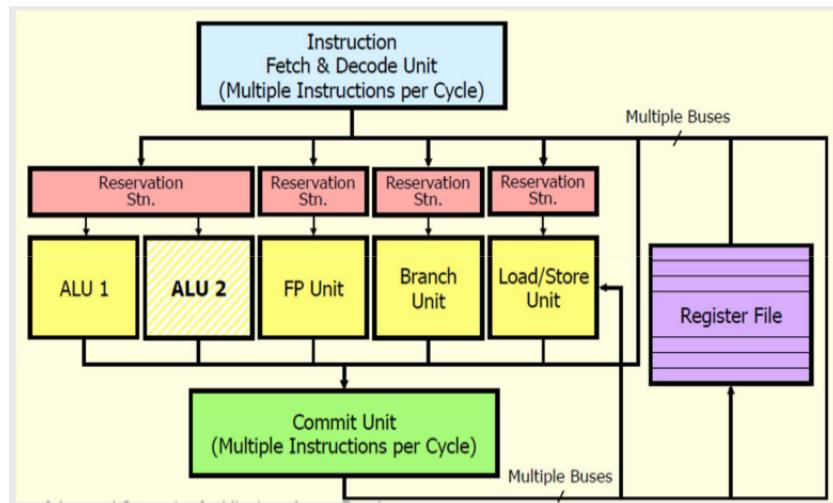
4.4 Esecuzione superscalare

È quello che fanno tutti i processori di fascia alta oggi (PowerPC, Pentium, Sparc,...)

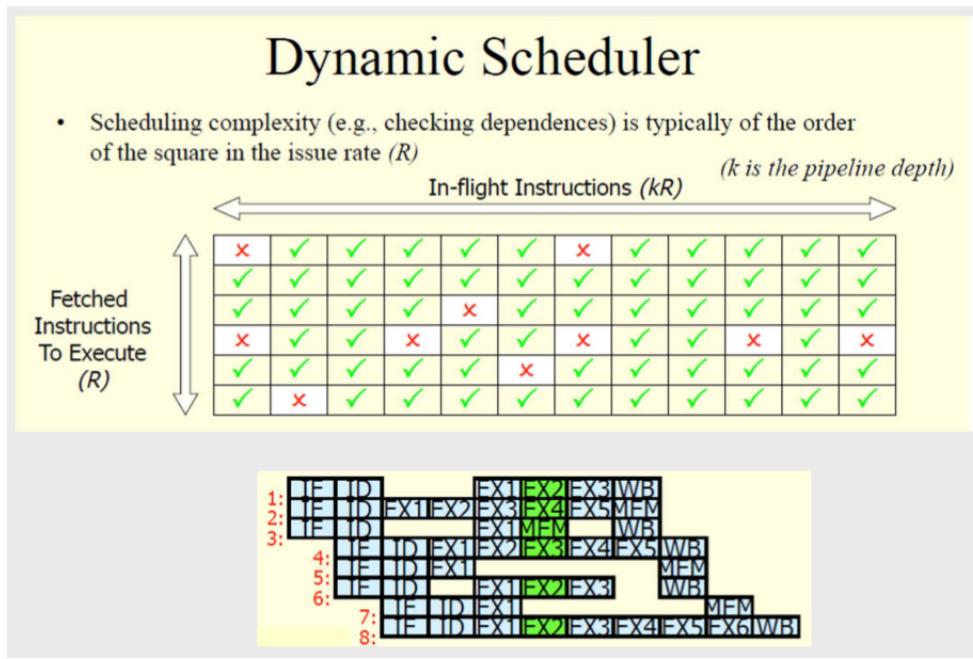
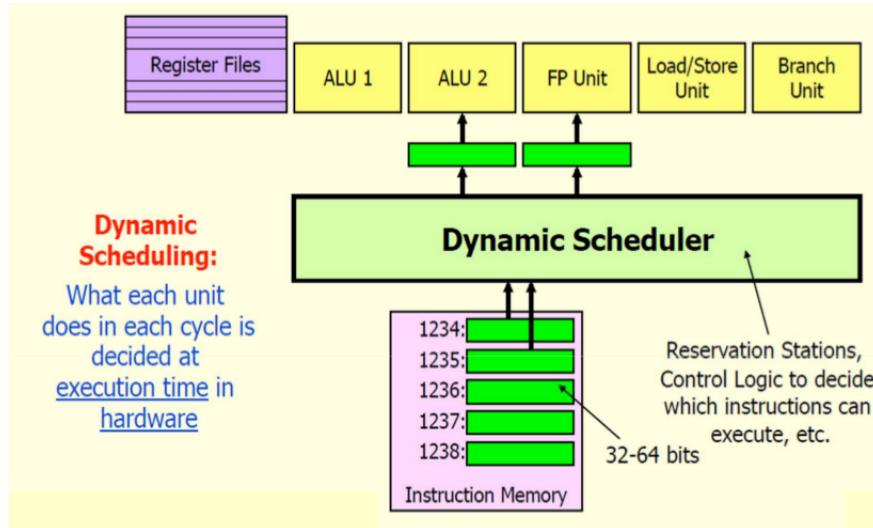
L'idea è iniziare l'esecuzione (**issue**) di **più di una istruzione per ciclo**.

I **requisiti** per farlo sono:

- Fare il fetch di più di una istruzione per ciclo: Non è difficile dato che la cache istruzioni può sostenere la banda.
- Decidere riguardo alla dipendenza dati e controllo: Lo scheduling dinamico si prende già cura di questo.



4.4.1 Struttura dello scheduler dinamico



- Ad ogni ciclo, il processore deve decidere quali istruzioni possono iniziare l'esecuzione.
- È necessario controllare tutte le istruzioni entrate in fetch con tutte le istruzioni "in volo", per capire quali sono indipendenti, prima di iniziare l'esecuzione.
- C'è un **limite** su quante istruzioni possono essere verificate durante un ciclo di clock.

4.4.2 Limiti

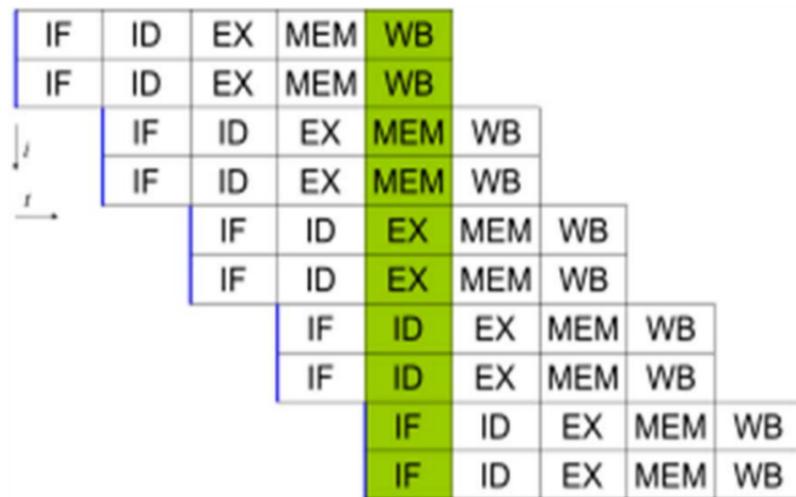
Lo scheduling dinamico costa!

- Abbiamo una quantità enorme di logica e un costo di area elevato.
 - Nel PowerPC 750 l'instruction sequencer è circa il 70% dell'area di tutte le unità di esecuzione
- Il tempo di ciclo è limitato dalla logica di scheduling (dispatcher e la logica di controllo associata).
- La verifica del design è molto complicata e la logica è complessa e irregolare.

Il numero di issue parallele è limitato nella pratica

- La **issue width** è il numero di istruzioni che possono essere messe in esecuzione in un singolo ciclo da un processore *multiple-issue* (chiamato anche ILP).

- Quando l'architettura superscalare fu inventata, furono realizzati processori a 2 e 4 issue



- Il massimo (raro) numero di issue parallele è 6
 - 2-issue: UltraSPARC-T2/T3, Cortex-A8, Cortex-A9, Atom, Bobcat
 - 3-issue: Pentium-Pro/II/III/M, Athlon, Pentium-4, Athlon 64/Phenom, Cortex-A15
 - 4-issue: UltraSPARC-III/IV, PowerPC G4e, Core 2, Core i, Core i2, Bulldozer
 - 5-issue: PowerPC G5
 - 6-issue: Itanium (ma è un processore VLIW)
- Questo perché è estremamente difficile trovare 8 o 16 istruzioni da eseguire in ogni ciclo (sono troppe!)
 - Richiede troppo tempo la computazione di scheduling (non riesce a restare in un ciclo di clock).
 - È necessario ridurre la frequenza del processore (guadagnando quindi in parallelismo ma perdendo in latenza e velocità).

4.4.3 Riassunto di processori superscalari e scheduling dinamico

In ogni caso abbiamo il vantaggio principale che il CPI ideale viene ridotto:

$$CPI_{ideal} = \frac{1}{\text{issue width}}$$

Svantaggi:

- È molto costosa la logica per trovare le dipendenze.
- Non scala: è complicatissimo aumentare la issue a più di 4 (e rischiamo di dover rallentare il clock).

4.5 ReOrder Buffer (ROB)

Al fine di ottimizzare le risorse hardware si consente alle istruzioni di procedere fuori ordine.

Assumendo che i conflitti RAW siano risolti, restano da gestire i conflitti WAR e WAW; in questi casi le istruzioni che causano conflitto devono restare in stallo prima di scrivere.

Il ReOrder Buffer è un componente che tiene traccia dei valori che le istruzioni hanno prodotto ma non ancora scritto sui registri. Questi valori restano nel buffer in attesa del completamento delle istruzioni che le precedono e nel frattempo sono resi disponibili per forwarding dei dati.

Questo consente:

- Riordino delle istruzioni.
- Gestione precisa delle eccezioni.
- Boosting.
 - Rollback sicuro in caso di errata speculazione.

- Disaccoppiare l'esecuzione delle istruzioni dal loro commit.

Soltamente si tratta di un buffer circolare con puntatore alla testa (prima entry libera) e alla coda (la prima in attesa di commit).

Di solito una entry del ROB contiene informazioni riguardo:

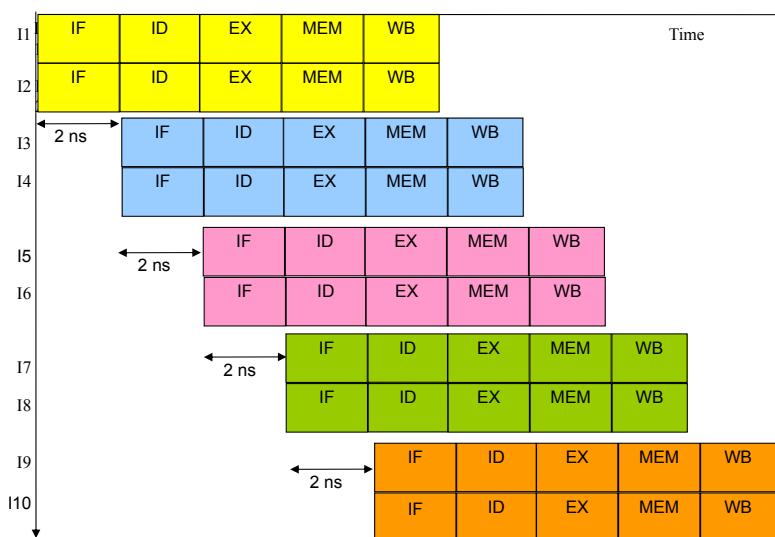
- PC dell'istruzione.
- Indirizzo del registro destinazione.
- Risultato.
- Stato dell'istruzione ed eventuali eccezioni.

5 Very Long Instruction Word (VLIW)

5.1 Idea di base

È un modo alternativo per ottenere l'*instruction level parallelism*.

- Aumenta le risorse hardware aumentando il numero di unità pipeline invece di complicare la stessa pipeline.
- Utilizza scheduling statico.
- Ogni unità in un ciclo esegue ciecamente quello che è già stato deciso a compile time.
 - Spostiamo dunque la complessità dall'hardware al compilatore.

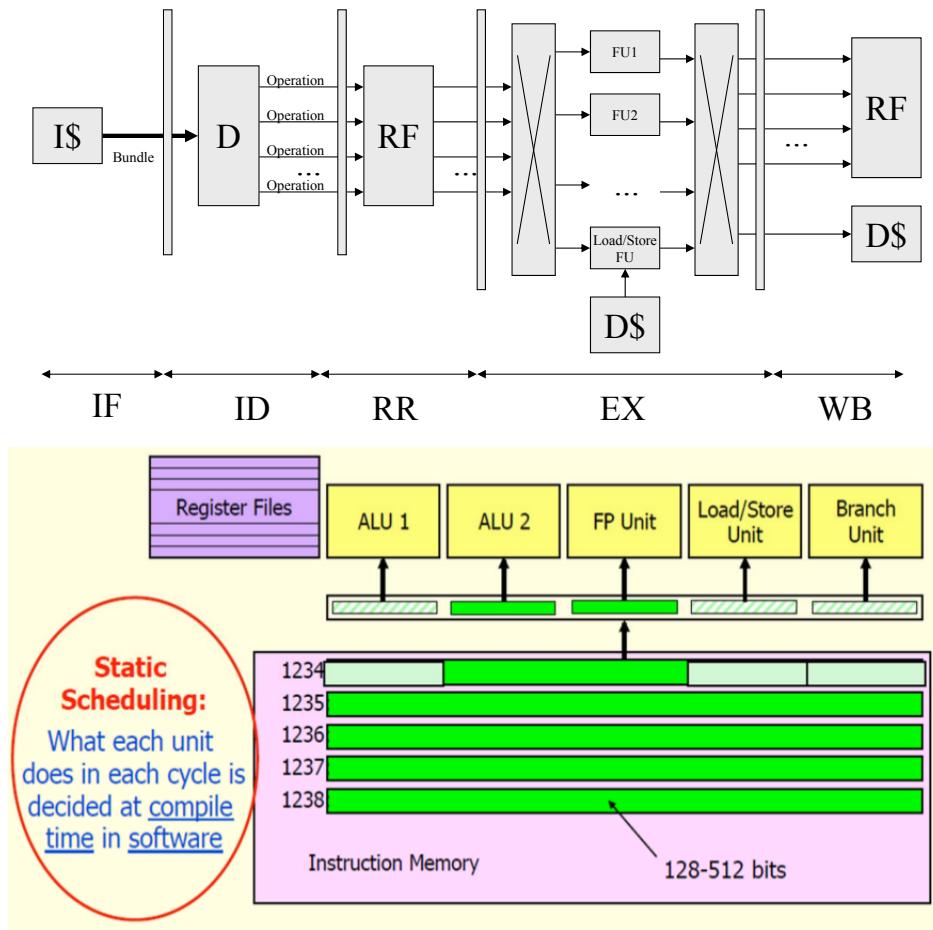


5.1.1 Come funziona

- Il compilatore raggruppa più istruzioni in un singolo *issue bundle*, anche detto **word**.
- Un bundle si compone di **slot**.
- Ciascuno slot del bundle viene riempito con un'istruzione, anche detta **sillaba**.
- Ad ogni slot corrisponde una fissata **unità funzionale (FU)** che si farà carico di eseguire quella sillaba.
- Le unità funzionali sono specializzate per eseguire una certa categoria di istruzioni: unità per operazioni ALU decimali, unità di branch, unità di load/store e così via.
- L'hardware eseguirà la fetch di una word ad ogni ciclo di clock.
- Il compilatore deve assicurarsi che non siano presenti *dipendenze* all'interno di un bundle o tra bundle vicini².
- L'hardware esegue le istruzioni fetchate
 - senza controllare la presenza di eventuali dipendenze.

²Nei processori "pure VLIW" non è ammessa la presenza di dipendenze tuttavia alcune implementazioni consentono al compilatore di inserire e segnalare una dipendenza. In questi casi la dipendenza verrà gestita dall'hardware attraverso tecniche di forwarding semplificate.

- senza dover scegliere a quale FU mandare l'istruzione caricata.



5.2 Caratteristiche

5.2.1 Limiti di VLIW

Parallelismo Il compilatore deve trovare molto parallelismo per mantenere le unità funzionali del processore occupate. Laddove non riesce a trovare sufficienti istruzioni per occupare la pipeline riempie il codice di NOP.

- Aumenta la dimensione del codice.

Per sfruttare parallelismo tra diversi basic blocks alcune implementazioni supportano una funzionalità chiamata *predication via select execution*.

- Il processore esegue entrambi i rami di una branch e quando il risultato (predicato) è noto decide quale dei due rami confermare e portare avanti.

Incompatibilità binaria Perdiamo la “portabilità” perché il processore è costretto a fare molte scelte strettamente dipendenti dalla micro-architettura. La compilazione infatti si basa su:

- ISA.
- Numero e tipo di FU disponibili.
- Latenza di ciascuna FU.

5.2.2 Vantaggi e svantaggi

Vantaggi:

- Scheduling statico a compile time:

- Grazie ad una visione globale del codice consente di sfruttare più parallelismo di quanto l'hardware non riuscisse a trovare.
- Non è limitato dal tempo di un ciclo di clock.

- L'hardware non deve effettuare alcun controllo su possibili dipendenze e hazard.
- Ogni unità funzionale è specializzata, la *decode* è molto semplice: i bit di ciascun campo sono noti a priori.
- La pipeline è più semplice:
 - si riduce l'area del chip (e quindi anche il prezzo).
 - si riduce il consumo di potenza.

Svantaggi:

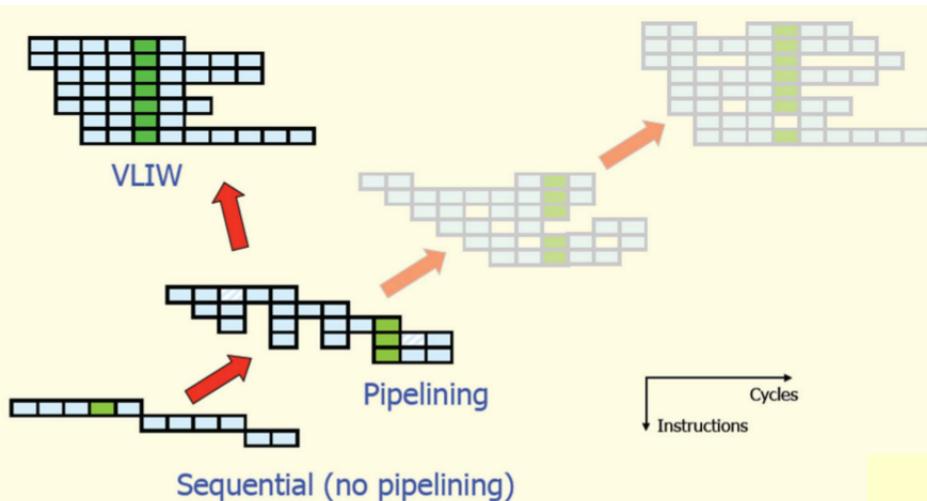
- Il compilatore deve fare tutto lui:
 - Si perde la portabilità del codice.
 - I compilatori sono molto più complessi.
- Invece di inserire stalli, vengono inserite NOP.
 - La dimensione del codice aumenta notevolmente.

5.2.3 Scheduling Software vs. Scheduling Hardware

- **Scheduling Software** (statico, fatto dal compilatore)
 - Codice sorgente disponibile.
 - Analisi “globali” possibili.
 - C’è più tempo disponibile perché non siamo dentro un ciclo di clock della CPU.
- **Scheduling Hardware** (dinamico, fatto dall’instruction scheduler)
 - Abbiamo informazioni a runtime (i dati attuali, indirizzi, puntatori, ...).
 - Portabilità del compilato.

5.2.4 Processori superscalari vs. VLIW

- I processori superscalari sono lo “stato dell’arte” commerciale per scopi generali (Intel Core i, Alpha, PowerPC, MIPS e Sparc sono superscalari).
- I processori VLIW hanno successo per dispositivi embedded (TriMedia media processors, C600 DSP, STMicroelectronics ST200 family, SHARC DSP, Itanium 2).



5.3 Scheduling VLIW

Come abbiamo visto finora il livello di parallelismo massimo raggiungibile è fortemente limitato. Le principali ragioni sono:

- Dipendenze tra le istruzioni.
- Dimensione ridotta del blocco basico.

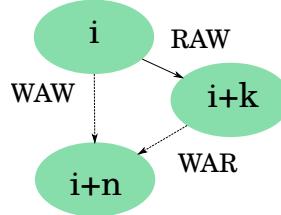
5.3.1 Grafo delle dipendenze

È possibile costruire un grafo delle dipendenze tra le istruzioni in modo da facilitare lo scheduling.

Dato il frammento di codice

```
i) A = B + 1;  
//codeblock  
i+k) X = A + C;  
//codeblock  
i+n) A = E + C;
```

possiamo ricavare il grafo



5.3.2 List based scheduling

Si basa sul grafo delle dipendenze e mira ad ottenere uno schedule *subottimo per un blocco basico*.

Prima di iniziare lo scheduling, si costruisca il grafo delle dipendenze. Si definisca inoltre il *ready set* come l'insieme di istruzioni pronte ad andare in esecuzione; si inizializzi il ready set con le istruzioni radice del grafo (quelle senza alcuna dipendenza). Inoltre ad ogni ciclo:

- schedulare istruzioni che:
 - sono contenute nel ready set.
 - a parità di altro, hanno priorità maggiore (percorso più lungo verso il fondo del grafo).
 - non impegnano una risorsa (FU) già impegnata in questo ciclo.
- Aggiungere al ready set tutte le istruzioni che hanno:
 - tutti i predecessori già schedulati.
 - tutti gli operandi pronti.

5.3.3 Loop unrolling

Per aumentare ulteriormente il parallelismo è necessario andare oltre la dimensione del blocco basico. Una tecnica non speculativa è il loop unrolling.

Il compilatore riconosce i cicli e, laddove è possibile determinare staticamente il numero di iterazioni del loop, srotola il ciclo. Quando viene srotolato un ciclo viene **aumentata la dimensione del blocco basico** formato istruzioni contenute nel ciclo **accorpando più iterazioni** tra loro indipendenti.

In un blocco basico di dimensione maggiore è possibile trovare maggiore parallelismo riordinando le istruzioni.

I problemi del loop unrolling

- dipendenze tra una iterazione e la successiva.
- terminazione del ciclo.

Possono esistere dipendenze tra le istruzioni di una iterazione e la successiva (**loop carried dependences**). In questo caso si può gestire le dipendenze introdotte oppure rinunciare ad eseguire l'unrolling.

La condizione di terminazione del ciclo deve essere adattata in modo da eseguire lo stesso numero di istruzioni per cui il ciclo era stato scritto. È possibile che srotolare il ciclo su base 4 (fondere 4 iterazioni assieme) possa portare ad avere **iterazioni residue** se l'originale numero di iterazioni non era multiplo di 4. In questo caso è necessario gestire come caso particolare le iterazioni residue accodandole al ciclo srotolato.

5.3.4 Loop peeling & fusion

Una tecnica simile allo srotolamento dei cicli è *sbucciare i cicli*. Questa tecnica consiste nell'**eliminare** un determinato numero di **iterazioni da un loop** aggiungendole poi in coda (o in testa) ad esso. Questo consente al compilatore di eseguire altre ottimizzazioni altrimenti non fattibili.

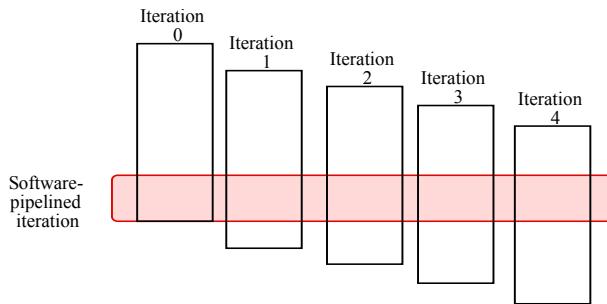
```
for (i = 0; i < 102; i++) b[i] = b[i + 2] + c; //loop A
for (j = 0; j < 100; j++) a[j] = a[j] * 2; //loop B
```

```
for (i = 0; i < 100; i++){
    b[i] = b[i + 2] + c;
    a[i] = a[j] * 2; } //fused loops
b[100] = b[102] + c;
b[101] = b[103] + c; // peeled from loop A
```

Nell'esempio qui sopra è possibile vedere come sbucciare due iterazioni dal primo loop consenta di fondere assieme i due loop. Il compilatore può inoltre decidere di srotolare il ciclo per aumentare ulteriormente la dimensione del blocco basico.

5.3.5 Software pipelining

Questa tecnica si applica a cicli nei quali vengono riconosciute istruzioni tra loro indipendenti inserite in diverse iterazioni.

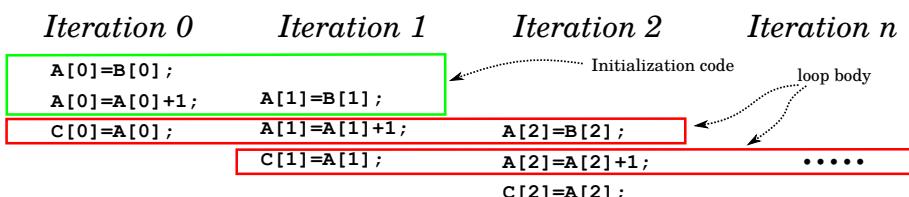


Se una situazione simile viene individuata allora è possibile riorganizzare il ciclo in modo che le istruzioni tra loro indipendenti facciano parte della stessa iterazione del nuovo ciclo.

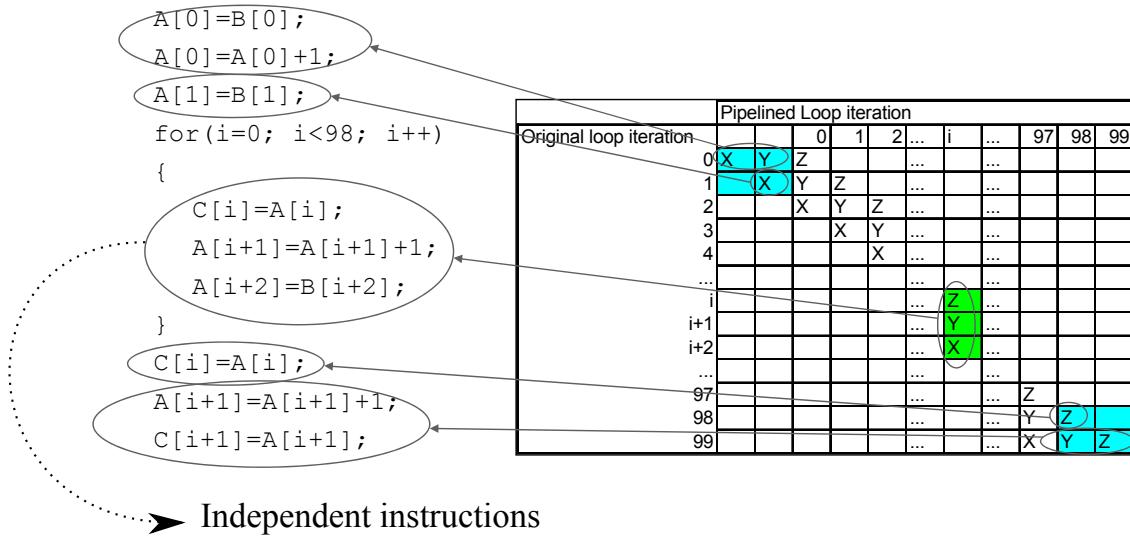
Per avere un esempio di applicazione si consideri il seguente frammento di codice.

```
for (i = 0; i < 100; i++) {
    A[i] = B[i]; //stage X
    A[i] = A[i+1]; //stage Y
    C[i] = A[i]; //stage Z
}
```

Analizzando le diverse iterazioni di questo ciclo possiamo osservare blocchi di istruzioni tra loro indipendenti a cavallo di iterazioni successive (nell'immagine seguente sono evidenziate in diversi riquadri).



Cercando una ricorrenza in queste indipendenze possiamo creare un nuovo loop riscrivendo in modo ottimizzato il codice dato.



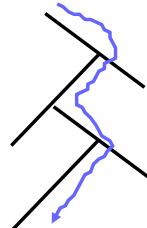
Si noti che in confronto al loop unrolling questa tecnica presenta il vantaggio di creare un binario più compatto (non necessita di duplicare il corpo del ciclo) e vengono gestite anche le dipendenze loop carried.

5.3.6 Trace scheduling

Trace scheduling è una tecnica di **global scheduling**, differente dalle tecniche viste finora che si limitavano ad eseguire un local scheduling (scheduling limitato a un blocco basico o a un ciclo). Fare global scheduling significa andare ad analizzare il software nel suo complesso e non limitandosi ad un blocco basico.

Trace scheduling si compone di due fasi:

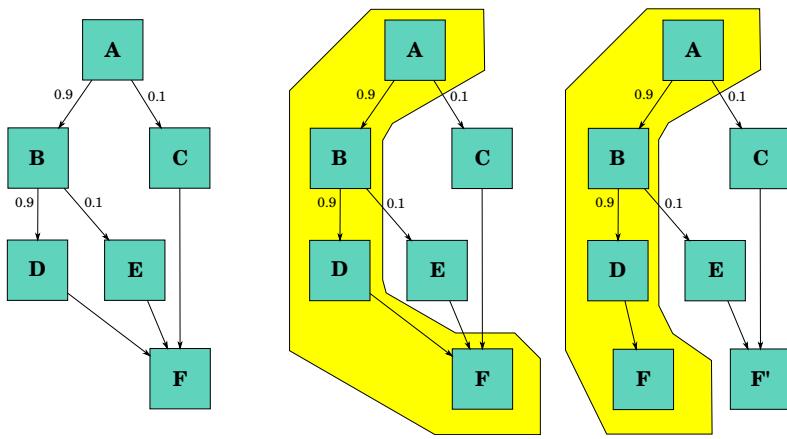
1. **Trace Selection.** In base al grafo di flusso del programma identifica la *traccia* di esecuzione più probabile. La sequenza di blocchi basici più probabile può essere determinata per:
 - predizione statica.
 - predizione per profiling.
2. **Trace Compaction.** Compatta la traccia individuata in un ristretto numero di istruzioni VLIW.
 - necessita di mantenere una procedura di *recovery* in caso di predizione mancata (necessita di altri registri in più).



5.3.7 Superblock scheduling

Evoluzione della tecnica trace scheduling. Si definisce **superblocco** un insieme di blocchi basici collegati con un singolo punto di ingresso e multiple uscite.

L'obiettivo è ottimizzare l'esecuzione del superblocco e trattare il resto del flusso di esecuzione come una eccezione. Il superblocco viene mantenuto il più compatto possibile, eventuali porzioni di codice con multipli ingressi vengono duplicate per mantenere la compattezza il più elevata possibile.



Vantaggi:

- ottimizzazione più semplice da realizzare.
- gestione delle eccezioni limitata alle uscite (in ingresso non ci sono mai problemi)

5.3.8 Predicated execution

Questa tecnica richiede una apposita estensione dell'ISA. Vengono introdotte *istruzioni condizionali* nel formato

(p) op Rd, Ra, Rb

dove p è un predicato o un registro a cui viene affiancata una normale operazione ternaria. Nella semantica delle istruzioni condizionali l'istruzione viene va in fase di commit solo se il predicato è true altrimenti viene trasformata in NOP.

L'introduzione di questo tipo di operazioni consente di tradurre costrutti di selezione in blocchi basici contenenti istruzioni condizionali.

Vantaggi:

- è possibile applicare riorganizzazione del codice in base alle dipendenze.
- non è necessario predire il risultato del branch per un if.
- la dimensione del blocco basico aumenta.

6 Confronto architetture multiple issue

Nome Architettura	Struttura Issue	Gestione Conflitti	Scheduling	Principali Caratteristiche	Esempi di Implementazione
Superscalare (statico)	Dinamica	Hardware	Statico	Esecuzione in order	sistemi embedded: MIPS, ARM (anche ARM Coretex A8)
Superscalare (dinamico)	Dinamica	Hardware	Dinamico	Esecuzione in parte out of order, nessuna speculazione	Nessuna al momento
Superscalare (speculativo)	Dinamica	Hardware	Dinamico con Speculazione	Esecuzione out of order con speculazione	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW / LIW	Statica	Principalmente Software	Statico	Tutti gli hazard sono identificati e segnalati (implicitamente) dal compilatore	sistemi signal processing: TI C6x
EPIC	Principalmente Statica	Principalmente Software	Per lo più Statico	Tutti gli hazard sono identificati e segnalati esplicitamente dal compilatore	Itanium

Parte III

Algoritmi e tecniche di dynamic scheduling

7 Punti cardine

7.1 Mantenere l'ordine

Mantenere l'ordine delle istruzioni in alcune fasi della pipeline semplifica molto tuttavia può anche rappresentare un collo di bottiglia. È necessario trovare un giusto bilanciamento.

7.1.1 Issue

Eseguire la issue **in order** consente di *conoscere il flusso di dati del programma* ovvero in che ordine le istruzioni accedono in lettura o scrittura ad un registro.

Con una issue out of order potremmo confondere conflitti WAR con conflitti RAW e viceversa.

Nel caso di multiple issue per ciclo di clock si tende a mantenere l'ordine anche se questo può rappresentare un limite significativo.

7.1.2 Execution

Eseguire le istruzioni in ordine semplifica il recupero degli operandi. Tuttavia è facilmente gestibile il caso di una esecuzione **out of order** con tecniche di scheduling dinamico. Questo consente di *gestire meglio i conflitti strutturali e aumentare il throughput*.

7.1.3 Commit

Il completamento delle istruzioni e scrittura su registro è facile se l'esecuzione è in order. **In order** commit significa *evitare conflitti WAR e WAW*. Il problema sorge nel caso in cui l'esecuzione sia out of order o le latenze di esecuzione sia variabile. In questi casi è necessario controllare la specifica assenza di conflitti o attendere il completamento di tutte le istruzioni precedenti.

Un commit in order consente inoltre una *gestione precisa delle eccezioni*, ovvero che tutte le istruzioni prima del PC hanno eseguito commit con successo e nessuna delle seguenti ha eseguito commit. In alcuni casi si considera accettabile una gestione non precisa delle eccezioni, in molti altri casi si preferisce adottare delle tecniche che consentano un in order commit.

8 Scoreboard

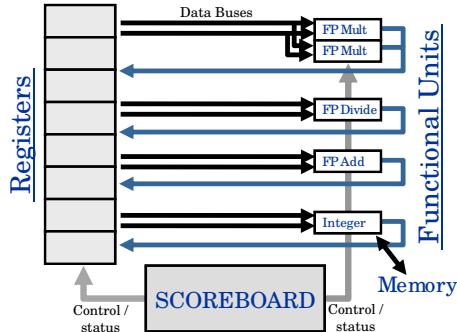
Per realizzare la funzionalità hardware di dynamic scheduling utilizziamo un algoritmo chiamato “**scoreboard dynamic scheduling algorithm**”

8.1 Caratteristiche generali della scoreboard

- La scoreboard rimpiazza gli stadi ID, EX, WB con 4 stadi:
 - Lo stadio **ID** viene suddiviso in:
 - * **Issue**: Decodifica le istruzioni, controlla gli hazards strutturali.
 - * **Read Operands** (aka **Read Register**): Aspetta fino a quando non ci sono data hazards, poi leggi gli operandi dai registri.
 - Una istruzione è nello stadio di **esecuzione** (EX) nell'intervallo di tempo tra quando *inizia* l'esecuzione e quando la *completa*. La scoreboard registra quindi:
 - * **Execution completed**: l'istante in cui l'esecuzione viene completata.
 - * **Write Result**: si accerta dell'assenza di hazard e finalizza il risultato.
- La **scoreboard** (tabella dei punteggi) permette alle istruzioni di essere eseguite se i primi due stadi sono pronti, senza aspettare altre istruzioni.
 - Le istruzioni vengono eseguite se non sono dipendenti da altre precedenti e non ci sono hazards.
 - Le istruzioni senza dipendenze vanno subito in esecuzione.
- Anche se l'issue è *in-order*, se abbiamo una lettura *out-of-order* degli operandi, avremo una esecuzione e completamento *out-of-order*.

- Tutte le istruzioni passano *in-order* attraverso lo stage ***issue***, ma possono andare in stallo o sorpassarsi a vicenda nello stadio ***read operands*** e quindi iniziare l'esecuzione *out-of-order* e (avendo latenze diverse) arrivare a completamento *out-of-order*.
- Assumiamo che la pipeline permetta l'esecuzione di più istruzioni allo stesso tempo. Abbiamo bisogno quindi di **unità funzionali multiple** o **unità funzionali in pipeline** (o entrambe).

- Nel processore CDC 6600 (il primo ad avere la tecnica di *dynamic scheduling*) la issue è *in-order*, l'esecuzione e il completamento sono *out-of-order*. Non c'è forwarding e abbiamo un modello impreciso per la gestione di interrupt ed eccezioni.



8.1.1 Hazard e dipendenze

Se il completamento può essere *out-of-order* significa che gli hazards di tipo **WAR** e **WAW** possono presentarsi.

- Per gli hazards **WAR** possiamo risolvere in due modi:
 - Generare uno **stall** sulla ***write back*** fino a quando i registri non sono stati letti.
 - Leggere i registri solo durante la fase ***read operands***.
- Per gli hazards **WAW** possiamo risolvere mettendo in stallo la issue di nuove istruzioni fino a quando l'altra istruzione non ha completato.

Inoltre:

- Non facciamo renaming dei registri.
- La scoreboard tiene traccia delle dipendenze e dello stato delle operazioni.
- **Hazard detection** e **resolution** viene centralizzata nella scoreboard:

1. Ogni istruzione passa per la scoreboard, dove viene costruito un record che tiene traccia delle dipendenze di quella istruzione.
2. La scoreboard determina se l'istruzione può leggere gli operandi e iniziare l'esecuzione.
3. Se la scoreboard decide che l'istruzione non può andare in esecuzione subito, controlla ogni modifica e decide quando l'istruzione può partire in esecuzione.
4. La scoreboard controlla quando l'istruzione può scrivere il suo risultato nel registro di destinazione.

8.1.2 Gestione delle eccezioni:

Il problema del completamento *out-of-order* è che dobbiamo preservare il comportamento delle eccezioni così come se fosse stato tutto *in-order*.

- La soluzione è quella di assicurarsi che non ci siano istruzioni che possono generare eccezione. Quando il processore trova che l'istruzione che lancia l'eccezione viene lanciata eseguirà le istruzioni critiche *in-order*.

Eccezioni “imprecise”: Una eccezione si dice “*imprecise*” se, quando viene sollevata, lo stato del processore non appare esattamente come quello di quando le istruzioni vengono eseguite *in-order*.

Possibili cause sono:

- La pipeline ha già completato istruzioni che sono **dopo** l'istruzione che causa l'eccezione nell'ordine corretto del programma.
- La pipeline non ha ancora completato istruzioni che sono **prima** l'istruzione che causa l'eccezione nell'ordine corretto del programma.

Le eccezioni “imprecise” rendono difficile riavviare l'esecuzione dopo aver sollevato l'eccezione (perché ci si trova in uno stato non previsto).

8.2 I 4 stadi di controllo della scoreboard

1. **Issue:** Decodifica l'istruzione e controlla la presenza di hazards strutturali e WAW hazards

- Le istruzioni vengono inviate nell'ordine del programma (per controllare gli hazards).
- Se una unità funzionale per l'istruzione è libera e non ci sono altre istruzioni con lo stesso registro destinazione (quindi no WAW hazard), la scoreboard fa issue delle istruzioni all'unità funzionale e aggiorna la sua struttura interna.
- Se un hazard strutturale o un hazard WAW esiste, allora l'issue va in stallo e non si fa issue di nessuna istruzione fino a quando l'hazard non è risolto
 - Notiamo che quando lo stage di issue va in stallo, il buffer tra lo stadio IF e Issue si riempie. Se il buffer ha una singola entry va in stallo tutto lo stadio IF, se il buffer è una coda allora lo stadio IF va in stallo quando la coda si riempie.

2. **Read Operands:** Attendi fino a quando non ci sono più hazard dati, poi leggi gli operandi

- Un operando sorgente è **disponibile** se:
 - Non c'erano istruzioni attive mandate in issue a scrivere quell'operando
 - Oppure una unità funzionale sta scrivendo il suo valore in un registro
- Quando l'operando sorgente è disponibile, la scoreboard dice all'unità funzionale di procedere a leggere gli operandi dai registri e iniziare l'esecuzione.
- Gli hazard **RAW** vengono **risolti dinamicamente** in questo stadio. Le istruzioni possono essere inviate in esecuzione *out-of-order*.
- Non c'è forwarding dei dati in questo modello.

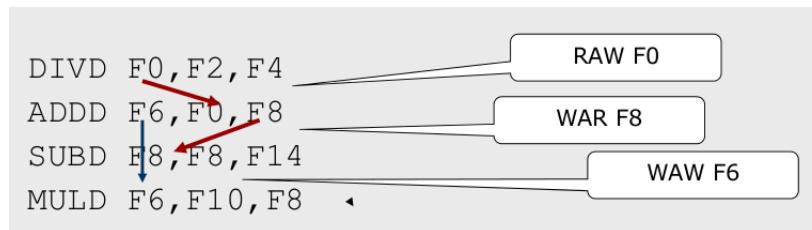
3. **Execution:** L'unità funzionale inizia l'esecuzione quando riceve gli operandi. Quando il risultato è pronto notifica la scoreboard che ha completato l'esecuzione.

- Le unità funzionali sono caratterizzate da:
 - **Latenza variabile** (il tempo effettivo utilizzato per completare una operazione)
 - Intervallo di inizializzazione (il numero di cicli che devono passare tra l'issue di due operazioni alla stessa unità funzionale).
 - La latenza delle operazioni di load/store dipende dalla data cache (HIT o MISS ?).

4. **Write result:** Controllare per hazards di tipo **WAR** e completare l'esecuzione

- Una volta che la scoreboard sa che l'unità funzionale ha completato l'esecuzione, essa stessa si preoccupa di **controllare** l'eventuale presenza di **hazards di tipo WAR**
 - Se non ce ne sono, scrive direttamente il risultato.
 - Se c'è un WAR, allora va in stallo prima di completare l'istruzione.

- Esempio:



- La scoreboard deve:
 - * Andare in stallo su SUBD nello stadio WB, aspettando che ADDD legga F0 e F8
 - * Inoltre deve andare in stallo su MULD nello stage di issue, fino a quando ADDD scrive F6
 - Si può anche risolvere utilizzando il **register renaming**.

8.3 Struttura della scoreboard

- Instruction status
 - Indica lo stato dell'unità funzionale (FU):
 - * Busy: L'unità è occupata o no
 - * Op: Indica l'operazione da effettuare con l'unità (+, -, ...)
 - * F_i : Indica il registro di destinazione
 - * F_j, F_k : Registri sorgente
 - * Q_j, Q_k : Unità funzionali che producono i registri sorgente F_j, F_k
 - * R_j, R_k : Flags che indicano quando F_j e F_k sono pronti. I Flags sono impostati su "no" dopo che gli operandi vengono letti
- Register result status: Indica quale unità funzionale scriverà quale registro. Non conterrà nulla se non ci sono istruzioni in coda che scriveranno quel registro

Instruction status	Wait until	Bookkeeping
Issue	Not busy (FU) and not result(D)	Busy(FU) \leftarrow yes; Op(FU) \leftarrow op; $F_i(FU) \leftarrow 'D'$; $F_j(FU) \leftarrow 'S1'$; $F_k(FU) \leftarrow 'S2'$; $Q_j \leftarrow \text{Result}'(S1)'$; $Q_k \leftarrow \text{Result}'(S2)'$; $R_j \leftarrow \text{not } Q_j$; $R_k \leftarrow \text{not } Q_k$; $\text{Result}'(D) \leftarrow FU$;
Read operands	R_j and R_k	$R_j \leftarrow \text{No}$; $R_k \leftarrow \text{No}$
Execution complete	Functional unit done	
Write result	$\forall f((F_j(f) \neq F_i(FU) \text{ or } R_j(f) = \text{No}) \text{ & } (F_k(f) \neq F_i(FU) \text{ or } R_k(f) = \text{No}))$	$\forall f(\text{if } Q_j(f) = \text{FU} \text{ then } R_j(f) \leftarrow \text{Yes})$; $\forall f(\text{if } Q_k(f) = \text{FU} \text{ then } R_k(f) \leftarrow \text{Yes})$; $\text{Result}(F_i(FU)) \leftarrow 0$; $\text{Busy}(FU) \leftarrow \text{No}$

8.4 Esempio di funzionamento della scoreboard

Struttura generale:

Instruction status:			Read	Exec	Write				
Instruction	j	k	Issue	Oper	Comp	Result			
LD	F6	34+	R2						
LD	F2	45+	R3						
MULTD	F0	F2	F4						
SUBD	F8	F6	F2						
DIVD	F10	F0	F6						
ADDD	F6	F8	F2						

Functional unit status:			dest	S1	S2	FU	FU	Fj?	Fk?
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Rj	Rk
	Integer	No							
	Mult1	No							
	Mult2	No							
	Add	No							
	Divide	No							

Register result status:									
Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
FU									

Passo 1 Supponiamo che il codice inizi l'esecuzione dalla prima LD senza hazard di alcun tipo. Al ciclo 1 possiamo eseguire la issue dell'istruzione. Scriviamo quindi 1 nella colonna Issue in corrispondenza della prima LD.

LD (Load Decimal) è una operazione che lavora su dati di tipo *Integer*. Verrà quindi occupata una FU per operazioni integer. Aggiorniamo quindi la tabella delle FU indicando che la FU Integer è *busy* e sta eseguendo una Load. Indichiamo inoltre i registri coinvolti (F6 destinazione, R2 registro sorgente 2). Il registro sorgente R2 è inoltre subito disponibile, riportiamo questa informazione nell'ultima colonna della tabella.

Infine procediamo ad aggiornare lo stato dei registri. Indichiamo che in F6 è stato "prenotato" per la scrittura dalla FU integer.

Instruction status:			Issue	Read	Exec	Write
Instruction	j	k	Issue	Oper	Comp	Result
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:			dest	S1	S2	FU	FU	Fj?	Fk?
Time	Name	Busy	Op	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	Yes	Load	F6	R2				Yes
	Mult1	No							
	Mult2	No							
	Add	No							
	Divide	No							

Register result status:			dest	S1	S2	FU	FU	Fj?	Fk?		
Clock			F0	F2	F4	F6	F8	F10	F12	...	F30
1											Integer

Passo 2 Al ciclo 2 è possibile iniziare la fase di Read Operands della prima LD. Scriviamo 2 nella seconda colonna della prima riga.

Non vi sono variazioni nello stato delle FU né nello stato dei registri.

La pipeline potrebbe proseguire con la issue della seconda LD tuttavia non sono disponibili risorse hardware per poter eseguire questa istruzione (l'unica FU integer è già occupata). La seconda LD deve quindi attendere. Si segnala quindi una issue stall dovuta a uno structural hazard.

La issue deve essere fatta in-order quindi le seguenti istruzioni non possono procedere.

Passo 3 La issue della seconda LD deve ancora aspettare per gli stessi motivi del ciclo precedente. Abbiamo una issue stall dovuta a uno structural hazard.

La prima LD esegue legge il valore da memoria (cache HIT) e termina la sua esecuzione. Indichiamo che il completamento della sua esecuzione avviene al ciclo 3 nella tabella. Le altre tabelle non subiscono variazioni.

Passo 4 La prima LD non ha problemi a eseguire la fase di write back. Riportiamo quindi 4 nella tabella dello stato delle istruzioni.

A questo punto l'istruzione è terminata e libera la FU. Puliamo la riga della FU integer prima di passare oltre.

La successiva LD non può ancora proseguire per consentire all'istruzione precedente di eseguire la fase di write back.

Passo 5 La issue della seconda LD può finalmente procedere. Indichiamo 5 nella colonna issue per la seconda LD. Come per il passo 1 riserviamo la FU integer per eseguire questa istruzione e aggiorniamo le altre tabelle.

Instruction status:			Issue	Read	Exec	Write
Instruction	j	k	Issue	Oper	Comp	Result
LD	F6	34+	R2	1	2	4
LD	F2	45+	R3	5		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

Functional unit status:			dest	S1	S2	FU	FU	Fj?	Fk?
Time	Name	Busy	Op	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	Yes	Load	F2	R3				Yes
	Mult1	No							
	Mult2	No							
	Add	No							
	Divide	No							

Register result status:			dest	S1	S2	FU	FU	Fj?	Fk?		
Clock			F0	F2	F4	F6	F8	F10	F12	...	F30
5											Integer

Passo 6 La seconda LD procede e inizia la fase di Read Operands.

L'istruzione successiva è una MULTD e possiamo mandarla in issue. Occupiamo quindi una FU moltiplicatore di quelle libere. Aggiorniamo la tabella delle FU riportando i dettagli dell'operazione di cui abbiamo fatto issue. Notare che questa operazione richiede due registri ma solo uno di questi è pronto. F2 infatti è il risultato della LD, ora in fase Read Operands (quindi non ancora terminata); possiamo notare che quando un registro non è pronto viene riportato anche l'id della FU che fornirà quel valore, per F2 si tratta della FU integer.

Aggiorniamo anche la tabella di stato dei registri indicando che F0 è un registro prenotato in scrittura dalla FU mult1.

Passo 7 La LD continua la sua esecuzione e compie un cache HIT completando la sua fase di esecuzione.

La MULTD ha un registro sorgente (F2) non ancora pronto e deve quindi attendere ancora. Abbiamo uno stallo della Read Operands dovuto a un *RAW hazard*.

L'istruzione successiva è una SUBD, eseguiamo la sua issue assegnandole la FU *add*. Aggiorniamo quindi le varie tabelle: nello stato delle FU possiamo vedere che l'operando F2 è marcato come non pronto; nello stato dei registri è riportato che F8 verrà fornito dalla FU *add*.

Instruction status:			Read	Exec	Write
Instruction	j	k	Issue	Oper	Comp Result
LD	F6	34+	R2	1	2 3 4
LD	F2	45+	R3	5	6 7
MULTD	F0	F2	F4	6	
SUBD	F8	F6	F2	7	
DIVD	F10	F0	F6		
ADDD	F6	F8	F2		

Functional unit status:			dest	S1	S2	FU	FU	Fj?	Fk?	
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	Yes	Load	F2			R3			Yes
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
	Divide	No								

Register result status:			F0	F2	F4	F6	F8	F10	F12	...	F30
Clock			FU	Mult1	Integer						
7											

Passo 8 La prossima istruzione del programma è una DIVD e non ci sono problemi nel caricarla nella FU *divide*. Avviene quindi la issue della DIVD.

Aggiorniamo le tabelle in seguito alla avvenuta issue.

La LD passa alla fase di write back senza problemi. Dopo aver scritto il registro F2 è necessario avvisare tutti coloro che erano in attesa di quel registro che il valore è pronto. Andiamo a cambiare lo stato di un po' di celle *ready* nella tabella dello stato delle FU.

Passo 9 - 10 Al passo 9 eseguono la fase di Read Operands le istruzioni MULTD e SUBD.

Al passo 10 le stesse iniziano l'esecuzione. Queste FU tuttavia hanno latenza tra loro diversa, maggiore di 1. È opportuno annotarsi ad ogni passo quanto manca al termine della fase di esecuzione per poter riportare il valore esatto in tabella.

La continuazione dell'esercizio è lasciata come facile esercizio al lettore.

Alla fine

Instruction status:			Read	Exec	Write
Instruction	j	k	Issue	Oper	Comp Result
LD	F6	34+	R2	1	2 3 4
LD	F2	45+	R3	5	6 7
MULTD	F0	F2	F4	6	19 20
SUBD	F8	F6	F2	7	9 11 12
DIVD	F10	F0	F6	8	21 61 62
ADDD	F6	F8	F2	13	14 16 22

Functional unit status:			dest	S1	S2	FU	FU	Fj?	Fk?	
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:			F0	F2	F4	F6	F8	F10	F12	...	F30
Clock			FU								
62											

8.5 Scoreboard con renaming esplicito

Al fine di ridurre l'attesa per la risoluzione dei conflitti WAW e WAR è possibile introdurre il renaming esplicito dei registri come ulteriore ottimizzazione.

- Non sono più necessari stalli in fase di commit.

Questa variante dell'algoritmo prevede l'utilizzo di un numero di registri superiore a quello specificato dall'ISA.

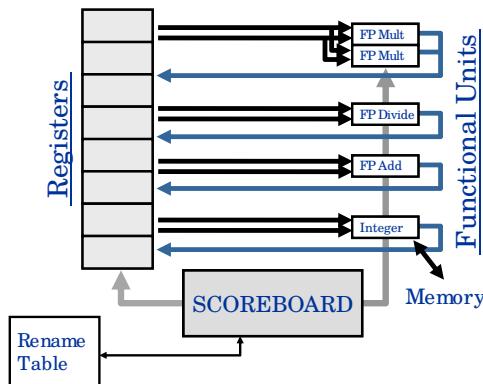
- Il programma conosce solo un sottoinsieme dei registri effettivamente disponibili (e usati).

L'idea di base è allocare un nuovo registro ogni volta che un'istruzione desidera scrivere nel RF.

- Si evitano conflitti WAR e WAR perché semplicemente non può accadere che due istruzioni scrivano sullo stesso registro fisico.

- Come la tecnica SSA, ma realizzata in hardware.

Viene mantenuta una lista di registri liberi. Un registro si dice libero quando non è ancora stato allocato; un registro torna ad essere libero quando nessuna istruzione attiva usa più quel registro.



8.5.1 Variazioni rispetto alla versione base

1. Issue

- Ad ogni issue, i registri vengono TUTTI rinominati attraverso la renaming table.
 - Al registro destinazione viene assegnato una nuova locazione libera.
 - Ai registri sorgente viene assegnato il nome del registro dove è (o sarà) locato il valore più recente del registro richiesto.
- Se non sono disponibili nuovi registri liberi si ha uno stallo sulla fase di issue per un conflitto strutturale.

2. Reorder buffer

- Non tiene traccia dei valori.
- Forza il commit in order.

3. Commit

- Scrive direttamente il valore nel registro.
- Libera i registri operandi se nessun altro li sta usando.

9 Algoritmo di Tomasulo

9.1 Caratteristiche principali

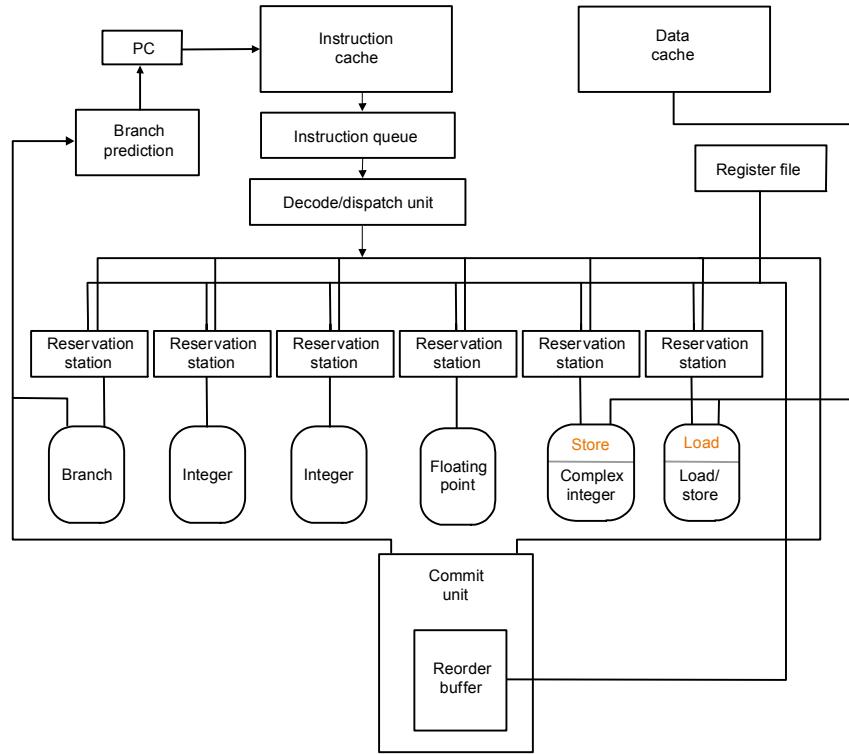
Si tratta di un altro algoritmo per effettuare scheduling dinamico. L'obiettivo è sempre quello di ottenere elevate performance senza necessità di compilatori particolari.

In particolare, rispetto a scoreboard:

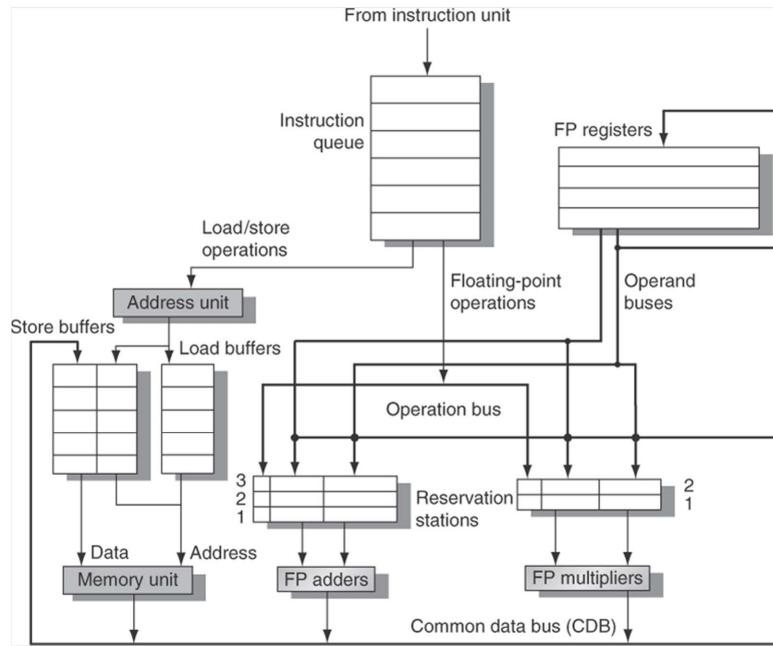
- I buffer sono distribuiti nelle varie FU e non centralizzati.
 - I buffer delle FU vengono chiamati **Reservation Station**.
- Nello stato delle istruzioni i registri vengono rimpiazzati dai valori o puntatori alle reservation station (RS) per consentire **register renaming**.
 - Evita così WAW e WAR hazard rinominando i risultati (usa numeri di RS invece di Register File).
 - Un numero di RS superiore al numero dei registri consente di effettuare ottimizzazioni che il compilatore non riesce a vedere.
- Utilizza il Common Data Bus invece dei registri per passare i valori da una FU ad un'altra (FU broadcast).
- Anche le istruzioni load e store hanno FU dedicate con RS.
- Separa istruzioni intere da istruzioni Floating Point (FP ops) su diverse code.

9.2 Schema architetturale

9.2.1 Schema generale di una architettura



9.2.2 Schema di una Floating Point Unit



9.2.3 Componenti di una Reservation Station

Tag identificatore della RS.

Busy indica se la RS è occupata o meno.

OP tipo di operazione da eseguire.

V_j, V_k valore degli operandi sorgente (per le load V_j contiene il valore di offset).

Q_j, Q_k puntatori alle RS che produrranno i valori V_j, V_k (0 indica che gli operandi sono già pronti).

Si noti che solo uno dei valori V o Q è significativo per un dato operando.

L'accesso in scrittura al register file non avviene mai direttamente per evitare conflitti.

Per le operazioni di Load e Store la RS ha un campo **Address** in cui viene memorizzato inizialmente l'offset (intero immediato) caricato dall'istruzione, poi l'effettivo indirizzo di memoria a cui l'operazione fa riferimento.

9.3 Gli stadi dell'algoritmo

9.3.1 Issue

- Preleva una istruzione dalla coda istruzioni (mantenuta FIFO)
 - viene garantita la **in-order issue**.
- Controlla se una RS è disponibile per il tipo di istruzione prelevato.
 - se non ce n'è una disponibile si verifica uno **structural hazard**.
 - * stallo.
 - se gli operandi richiesti non sono subito disponibili si tiene traccia di quali FU produrranno quei valori tramite i puntatori Q .
- Rinomina i registri
 - Risoluzione dei **WAR**, **WAW**: la issue di una istruzione che legge (o scrive) un registro destinazione di un'altra istruzione già caricata salverà il riferimento alla RS dell'istruzione che già occupa il registro in scrittura.

9.3.2 Execution

- **La fase di esecuzione inizia quando entrambi gli operandi sono pronti e una FU è disponibile.**
- Se gli operandi non sono pronti **controlla il Common Data Bus** (CDB) in attesa degli stessi.
- Ritardando l'esecuzione fino a quando gli operandi non sono pronti si evitano i RAW.
 - Si noti che gli stalli dovuti ai conflitti RAW hanno durata inferiore in quanto il dato quando viene pronto non deve aspettare la write back sul Register File ma giunge alla RS tramite il CDB (è una specie di Forwarding).
- Prima di mandare in esecuzione una istruzione pronta è necessario verificare che sia disponibile una FU.
 - A causa degli stalli, è possibile che più di una istruzione diventi pronta per essere eseguita nello stesso ciclo di clock causando conflitto strutturale.

Gestione dei Branch Nessuna istruzione può iniziare la fase di esecuzione prima che il risultato di tutti precedenti branch sia noto.

Nel caso di branch prediction deve essere noto il risultato definitivo.

Esecuzione di una Load / Store Avviene in due passi:

1. Calcola l'effettivo indirizzo target.
 - (a) Aspetta che il valore del registro base sia disponibile.
 - (b) Calcola il valore e lo piazza nel load (o store) buffer.
2. Invio all'unità di memoria:
 - La load nel load buffer viene eseguita appena una unità di memoria è disponibile.
 - La store nello store buffer aspetta che il valore da scrivere venga pronto prima di essere inviata all'unità di memoria.

Conflitti di memoria Tramite il calcolo preventivo dell'indirizzo target è possibile evitare conflitti di memoria.

- Load e store possono anche essere eseguite in ordine qualsiasi, purché accedano a diversi indirizzi di memoria.
- È necessario che prima di ogni operazione di accesso a memoria siano calcolati gli indirizzi target di tutte le operazioni di memoria precedenti.
 - Viene controllata la presenza di un indirizzo target identico in istruzioni precedenti.
 - * Ogni load che trova un indirizzo target identico in un write buffer attende.
 - * Ogni write che trova un indirizzo target identico in un write o load buffer attende.

9.3.3 Write Result

Quando il risultato è disponibile si passa nella fase di write result.

- il risultato viene scritto sul CDB.
- Dal CDB il risultato viene scritto sul Register File.
- Dal CDB il risultato viene scritto in tutte le RS in attesa di quel dato.
- Le istruzioni store scrivono il dato in memoria.

Al termine di tutto ciò le RS liberate vengono marcate disponibili.

9.4 Altri dettagli

9.4.1 Common Data Bus

Il common data bus è un bus su cui viaggia una componente dati e una componente che identifica la sorgente.

- nel IBM 360/91 le dimensioni erano 64 bit dati + 4 bit sorgente.

Ogni FU deve eseguire una ricerca associativa nelle RS sui 4 bit di sorgente per verificare se il dato sul CDB è di suo interesse o meno.

Ogni RS deve avere un buffer associativo molto veloce.

Un singolo CDB potrebbe essere un collo di bottiglia.

9.4.2 Confronto Tomasulo vs Scoreboard

- Le fasi
 - Tomasulo:
 - * Issue (in order)
 - * Start Execution (out of order)
 - * Write Result (out of order)
 - Scoreboard:
 - * Issue (in order)
 - * Read Operands (out of order)
 - * Execution (out of order)
 - * Write Result (out of order)
- Conflitti strutturali
 - Tomasulo: in base alla disponibilità di RS
 - Scoreboard: in base alla disponibilità di FU
- Risoluzione conflitti WAR, WAW
 - Tomasulo: renaming隐式通过RS
 - Scoreboard: inserisce stalli prima del completamento
- Propagazione dei risultati
 - Tomasulo: broadcast tramite CDB
 - Scoreboard: scrive direttamente sul Register File, è possibile introdurre il forwarding
- Controllo
 - Tomasulo: distribuito nelle varie RS
 - Scoreboard: centralizzato attraverso la scoreboard
- Inoltre

- Tomasulo esegue implicitamente loop unrolling via hardware con un effetto paragonabile a quello della tecnica SSA (Single Static Assignment)
- Tomasulo aumenta notevolmente la complessità hardware rispetto a Scoreboard
- In entrambi gli algoritmi i limiti del parallelismo sono
 - * i branch
 - * la gestione delle eccezioni

imprecisa non è garantita l'integrità degli operandi per l'handler di eccezioni
inesatta le modifiche apportate dall'handler vengono viste dalle istruzioni solo dopo l'eccezione.

9.5 Esempio di funzionamento

Passo 1

Instruction status			Start	Write			
Instruction	j	k	Issue	Execute	Result		
LD F6	34+	R2	1				
LD F2	45+	R3					
MULTFO	F2	F4					
SUBF8	F6	F2					
DIVF10	F0	F6					
ADD F6	F8	F2					

Load1	v1	q1	v2	q2			
Load2	34		v(R2)				
EXLOAD							

add1	v1	q1	v2	q2			
add2							
EXADD							

mult1	v1	q1	v2	q2			
mult2							
EXMUL							

RF	0	1	2	3	4	5	6	7	8	9	10	11	12
Q						Load1							

Come vediamo dall'immagine, all'avvio parte la issue della LD. Indichiamo lo stato della prima istruzione scrivendo che la issue inizia al ciclo 1.

In contemporanea viene occupata una RS della FU Load: il primo operando, costante, viene copiato nella Reservation Station. Il secondo operando è un registro; il registro è subito disponibile, viene quindi letto e il suo valore copiato nella Reservation Station.

Rimane da aggiornare lo stato del Register File. Questa prima istruzione scriverà su F6: riportiamo in corrispondenza del registro F6 l'identificativo della FU che produrrà tale valore.

Passo 2 Al passo successivo la prima istruzione verifica che tutti i suoi operandi sono pronti nella RS e inizia la fase di esecuzione. Si riporta questa informazione in tabella.

L'istruzione successiva è anch'essa una LD. È disponibile un'altra RS per la FU Load, viene quindi occupata per eseguire la issue di questa seconda istruzione. Riportiamo sullo stato delle istruzioni che al ciclo 2 si esegue la issue della seconda istruzione. La RS viene riempita con i valori: una costante e il valore di un registro già pronto.

Aggiorniamo inoltre lo stato del RF indicando che il registro F2 verrà scritto dalla FU appena allocata.

La prima istruzione, entrata in fase di esecuzione, calcola l'indirizzo effettivo di memoria e lo scrive nel load buffer.

Instruction status			Start	Write			
Instruction	j	k	Issue	Execute	Result		
LD F6	34+	R2	1	2			
LD F2	45+	R3	2				
MULTFO	F2	F4					
SUBF8	F6	F2					
DIVF10	F0	F6					
ADD F6	F8	F2					

Load1	v1	q1	v2	q2			
Load2	34		v(R3)				
EXLOAD	45		v(R2)				

add1	v1	q1	v2	q2			
add2							
EXADD							

mult1	v1	q1	v2	q2			
mult2							
EXMUL							

RF	0	1	2	3	4	5	6	7	8	9	10	11	12
Q			Load1										

Passo 3 Al ciclo 3 la prima istruzione prosegue la fase di esecuzione (accesso a memoria) mentre la seconda rimane in attesa della prima per poter accedere al load buffer, quindi alla memoria.

L'istruzione successiva è una MULTD, una RS della FU adatta è libera e si procede alla fase di issue. Occupiamo una RS e indichiamo l'avvenuta issue in tabella. Indichiamo inoltre che il registro destinazione di quest'operazione sarà prodotto dalla RS appena occupata.

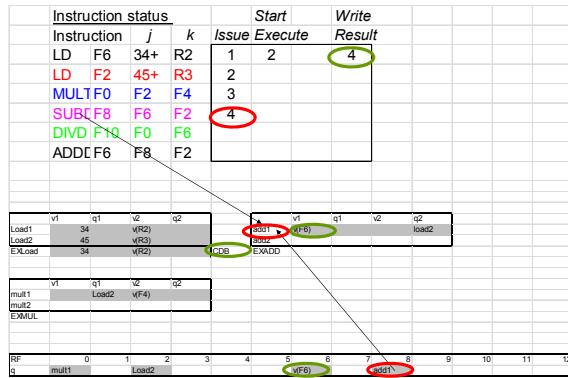
Passo 4 La prima istruzione termina l'accesso a memoria e scrive i risultati: indichiamo in tabella che questi ultimi vengono scritti al ciclo 4.

La seconda operazione deve attendere che la prima liberi il buffer e attende anche per questo ciclo. Si ricorda che l'esecuzione di una LD termina quando i dati sono stati scritti.

La MULTD necessita del registro destinazione della seconda LD e attende fino a quando questo non sarà pronto.

L'istruzione successiva è una SUBD e viene caricata nella prima RS della FU per le somme. Riportiamo in tabella l'avvenuta issue, l'occupazione della RS e nel RF status che il registro destinazione sarà prodotto dalla RS appena occupata.

In questo ciclo nel CDB è disponibile il valore del registro da scrivere, risultato della prima operazione; aggiorniamo anche il RF e le RS che attendevano questo dato (nello specifico la RS della SUBD), in questo caso il CDB viene usato per effettuare una sorta di forwarding del dato.



Passo 5 Al ciclo 5 la seconda LD inizia la fase di esecuzione e occupa il load buffer. Le successive due istruzioni sono in attesa di un operando e quindi non possono procedere oltre.

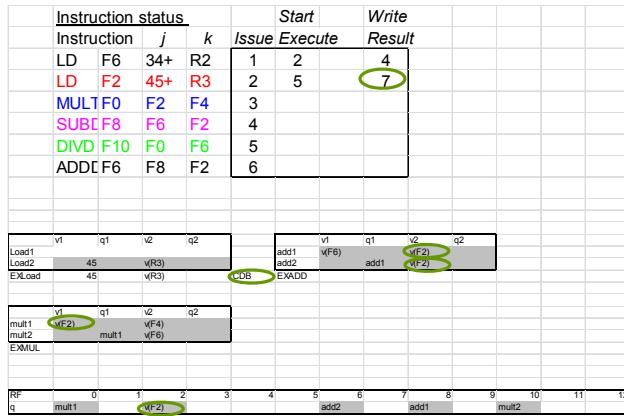
Avviene la fase di issue della successiva istruzione, una DIVD che occupa la seconda RS della FU per le moltiplicazioni. Aggiorniamo lo stato della RS e indichiamo nel RF status che il registro destinazione sarà fornito da questa RS.

Passo 6 Al ciclo 6 la seconda LD prosegue la fase di esecuzione; le successive istruzioni rimangono in attesa di operandi mancanti.

Avviene la issue dell'ultima istruzione. La ADDD viene caricata nella RS libera della FU per le addizioni. Si noti come il registro destinazione di questa operazione sia anche registro sorgente di una precedente istruzione che ancora deve essere eseguita. Quest'ultima ha però già effettuato la fase di issue e ha letto il valore precedentemente contenuto nel registro; il renaming implicito ha in questo caso eliminato il conflitto WAR che si poteva presentare.

Passo 7 Al ciclo 7 la seconda LD termina l'accesso a memoria e scrive il valore sul CDB. Il valore viene quindi copiato in tutte le RS delle operazioni che richiedevano quel valore.

Le altre istruzioni ancora non possono partire in esecuzione.



Passo 8 Al ciclo 8 le FU di moltiplicazione e addizione portano ciascuna in esecuzione una delle loro operazioni, la MULTD e la SUBD. La scelta è forzata in quanto sono le uniche ad avere tutti gli operandi pronti.

Passi successivi Supponendo che per ogni addizione la latenza di esecuzione sia di 2 cicli di clock, 10 cicli per ogni moltiplicazione e 40 cicli per la divisione, al termine dell'esecuzione si avrà la situazione seguente.

Instruction status			Start	Write			
Instruction	j	k	Issue	Execute	Result		
LD F6	34+	R2	1	2	4		
LD F2	45+	R3	2	5	7		
MULTF0	F2	F4	3	8	18		
SUBF8 F6	F2		4	8	10		
DIVF10 F0	F0	F6	5	19	59		
ADD F6 F8	F8	F2	6	11	13		

v1 q1 v2 q2	v1 q1 v2 q2
Load1 Load2 EXLoad	add1 add2 EXADD
v1 q1 v2 q2	v1 q1 v2 q2
mult1 mult2 vF0 vF6 EXMUL vF0 vF6	DB
RF 0 vF0 1 vF2 2 vF6 3 vF8 4 vF10 5 vF12 6 vF14 7 vF16 8 vF18 9 vF20 10 vF22 11 vF24 12 vF26	

9.6 Variante speculativa

9.6.1 Tomasulo con ROB

In questa variante i risultati non vengono più scritti alla fase di commit ma passano da un ReOrder Buffer. Si ha **in order commit**.

In caso di RAW, una istruzione memorizza l'indirizzo del ROB invece dell'indirizzo della RS.

Nel ROB viene allocata una entry per ogni istruzione quando questa viene caricata (fase di issue).

9.6.2 Le fasi di Tomasulo speculativo

1. Issue
 - Scrive il dato sul CDB e sul ROB.
 - Le RS in attesa del valore prodotto lo leggono.
 - Libera la RS.
2. Execution
3. Write Result
 - Commit Normale: il risultato dell'istruzione più vecchia è pronto nel ROB e viene scritto nel RF.
 - Store Commit: come prima, ma la scrittura avviene in memoria.
 - Rollback: in caso di branch con predizione non corretta il ROB viene completamente svuotato.
4. Commit

9.6.3 Contenuto del ROB

Tipo di istruzione	Destinazione	Valore	Ready
--------------------	--------------	--------	-------

- Tipo di Istruzione:
 - ALU / Load
 - Store
 - Branch
- Destinazione:
 - Registro destinazione (ALU / Load)
 - Indirizzo destinazione (Store)
 - vuoto (Branch)

9.6.4 Altre variazioni

Il caso della store Il ROB rimpiazza completamente lo store buffer. La store viene eseguita in due fasi, l'accesso a memoria avviene durante la fase di commit tramite il ROB.

Renaming Il renaming non viene più fornito dalle RS ma dal ROB. Tutti i puntatori puntano alle entry del ROB invece che alle RS.

Le eccezioni Una eccezione non viene raccolta fino a quando non viene eseguito il commit dell'istruzione associata. Questo garantisce una gestione precisa delle eccezioni.

Parte IV

Oltre ILP

10 TLP e DLP

DLP Data Level Parallelism. Eseguire in parallelo la stessa istruzione su array di dati tra loro indipendenti.

TLP Thread Level Parallelism. Sfruttare più flussi di istruzioni indipendenti alla volta (multithreading).

10.1 Multithreading

Funziona bene quando la quantità di parallelismo intrinseco nel programma diventa collo di bottiglia.

Il multithreading riduce il tempo di esecuzione più di quanto non aumenti il consumo energetico.

Una **caratteristica importante** del *multithreading* è che ci permette di ridurre il tempo di esecuzione più di quanto aumenti la potenza consumata (a differenza delle tecniche di *Instruction Level Parallelism*).

10.1.1 I thread

Vengono creati:

- Esplicitamente dal programmatore.
- Implicitamente dal sistema operativo.

Granularità quantità di computazione assegnata ad un thread.

- Thread a granularità **fine**.
 - contengono solo alcune istruzioni.
 - ottimali per *multithreading uniprocessore* (1 processore, n thread).
- Thread a granularità **grossolana**.
 - contengono centinaia o migliaia di istruzioni.
 - ottimali per *multithreading multiprocessore*.

10.1.2 Cosa si introduce

- Multipli PC
- Multipli RF
- Memoria Virtuale
- Thread context switch
 - rapido e più veloce del process context switch

Non si replicano le FU. Il multithreading consente di condividere le FU di uno stesso processore tra diversi thread. Si suppone comunque che il processore condiviso abbia diverse FU che possono lavorare in parallelo.

10.2 Tipologie di TLP

10.2.1 Multithreading a granularità grossolana.

Quando un thread si ferma per uno stallo, un altro thread viene eseguito.

- Lunghi stalli (per esempio cache miss) vengono nascosti dal context switch.
 - ad ogni context switch è necessario attendere lo svuotamento della pipeline.
 - dopo ogni context switch ci sono alcuni cicli idle di penalità di ripresa.
- Conviene quando $\text{stall}_{\text{time}} \gg \text{pipeline refill}_{\text{time}}$

10.2.2 Multithreading a granularità fine.

Si passa da un thread all'altro ad ogni istruzione.

- Politica round robin per decidere da quale thread fare issue.
 - ad eccezione dei thread in stallo (per evitare 0 issue).
- Rallenta l'esecuzione dei singoli n thread ma aumenta il throughput.
 - $CPI_{thread} = n \times$
 - $CPI_{core} = 1 \times$
- Essendo i core molto semplici, è facile combinarli in una architettura multicore.
 - Solitamente ogni core ha una architettura single issue o 2issue.

10.2.3 Simultaneous multithreading.

Diversi thread mandano istruzioni in issue nello stesso ciclo di clock.

- SMT = TLP + ILP
- Tramite *register renaming* si riesce a non mischiare i dati.
- La risoluzione delle dipendenze è lasciata ad uno scheduler dinamico.
- Cerca di sfruttare al massimo le risorse hardware a disposizione.
 - Tipologia di TLP maggiormente diffusa.
 - Processori di questo tipo riescono ad ottenere buoni speedup ad efficienza energetica pressoché invariata.

Necessita di una unità di fetch migliorata, infatti deve:

- Caricare istruzioni da più thread.
- Scegliere da quale thread caricare.

Possono essere combinati più core SMT in un unico processore.

- la tecnologia *Intel hyperthreading* funziona esattamente così.

11 Multiprocessori

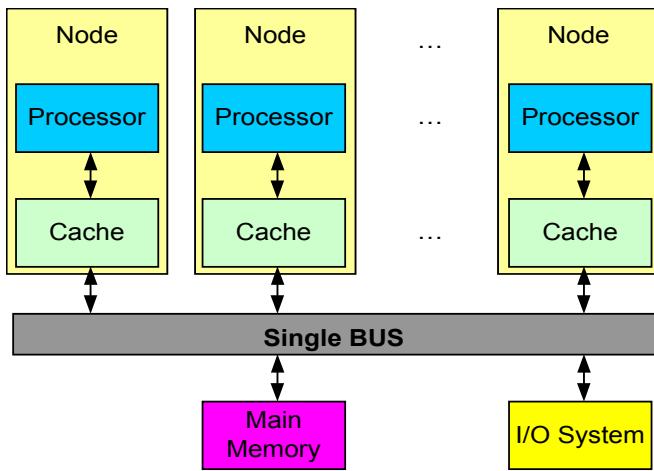
Con il termine multicore si intendono computer con poche coppie di processori, a differenza dei manycore che contengono centinaia o più core per ogni processore.

11.1 Topologie di connessione

11.1.1 Single bus vs network

Single bus

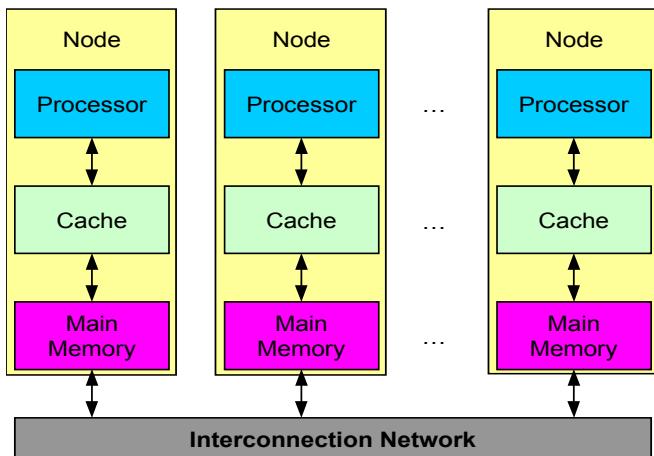
- Limite numero di processori connessi su singolo chip.
 - Ad oggi, in una architettura a single bus si è raggiunto il massimo di 36 processori su un singolo chipset.
- Accesso al bus per ogni operazione di accesso a **memoria**.
 - Abbiamo bisogno **tanta** banda per trasferire i dati.



- È una soluzione “economica”

Network connected

- Ogni processore ha una memoria interna.
- Il bus viene usato solo per comunicare tra processori, non far transitare i dati dalla memoria.
- Diverse topologie (non necessariamente identiche per bus dati e bus indirizzi)



È una soluzione più costosa, a parità di performance. Il costo della soluzione dipende da quanto vogliamo rendere interconnessi i vari processori tra loro. Collegarli tutti tra di loro è costoso, usare un unico bus è economico ma lento, quindi possiamo trovare delle vie di mezzo.

11.1.2 Varie topologie di connessione

- Bus
- Ring
- Mesh
- N-cube
- Crossbar Network
- Thorus

Possono inoltre esserci bus separati per gli indirizzi e per la memoria.

11.2 Confronto delle topologie di network-connected multiprocessors

Rappresentiamo le reti di multiprocessori utilizzando:

- **Nodi:** Sono dei nodi processore-memoria
- **Switch:** Collega i processori-memoria e altri switch
- **Archi:** Rappresentano un link di collegamento (bidirezionale)

Costi Il costo di una topologia si misura in termini di:

- Numero di **switch**
- Numero di **link per switch**
- Dimensione del link (**larghezza in bit**)
- Lunghezza dei link in hardware

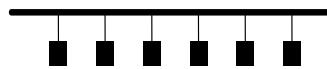
Metriche Dati:

- P = numero dei nodi
- b = banda di un singolo link

Calcoliamo:

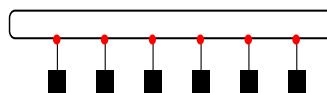
- **Banda totale** (caso ottimo)
 - È la banda di ogni link moltiplicata per il numero di links.
- **Banda di bisezione** (caso pessimo)
 - Viene calcolata dividendo la macchina in due parti, ciascuna con metà dei nodi. Calcoliamo poi la somma della banda dei links che attraversano quella linea immaginaria.

11.2.1 Singolo Bus



- Banda totale = b
- Banda di bisezione = b

11.2.2 Ring

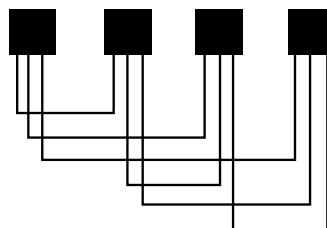


- Banda totale = $P \cdot b$
- Banda di bisezione = $2 \cdot b$

L'architettura a ring è meglio di quella a bus sicuramente, però nel caso pessimo (quello della banda di bisezione) abbiamo uno speedup solo di un fattore 2.

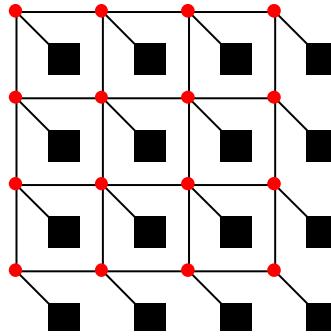
11.2.3 Crossbar network

Ogni processore ha un link bidirezionale verso gli altri, senza switch in mezzo.



- Banda totale $\frac{(p \cdot (p-1))}{2} \cdot b$
- Banda di bisezione $(\frac{p}{2})^2 \cdot b$

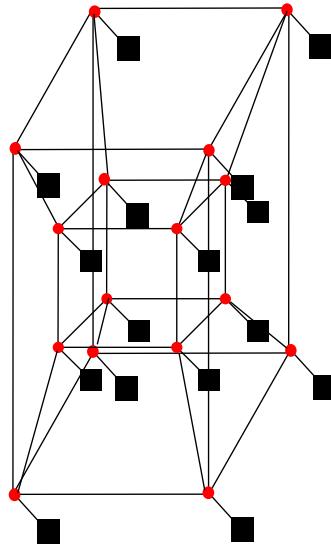
11.2.4 2D Mesh



sia $N = \sqrt{p}$

- Banda totale $2N(N - 1) \cdot b$
- Banda di bisezione $N \cdot b$

11.2.5 Ipercubo



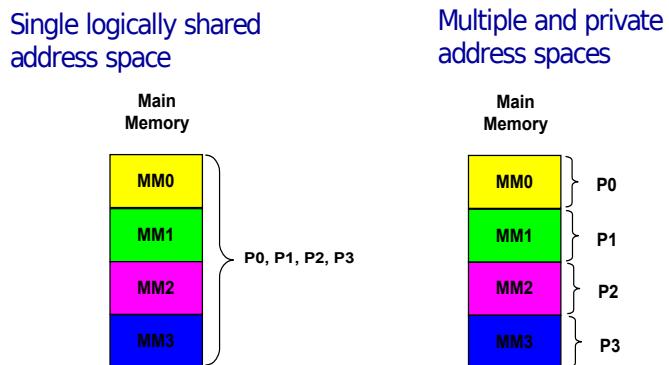
sia $N = n^o$ di nodi vicini

- Banda totale $\frac{N \cdot p}{2} \cdot b$
- Banda di bisezione $2^{N-1} \cdot b$

11.3 Modello di indirizzamento della memoria

Come i multiprocessori condividono dati.

11.3.1 Organizzazione logica



- Single, logically shared address space:

- Ogni processore può fare riferimento ad un dato in un punto qualunque della memoria.
- I processori comunicano tra di loro tramite dati condivisi sulla memoria
- “Shared Memory” non significa che c’è un’unica memoria. La memoria potrebbe anche essere fisicamente distribuita tra i vari processori.
 - * Nel caso in cui sia distribuita abbiamo anche il problema di gestire i meccanismi di coerenza della cache tra i vari processori

- **Multiple and private address spaces:**

- I processori comunicano tra di loro tramite un meccanismo di send/receive (si parla di **Message Passing Architecture**)
- Lo spazio di indirizzamento è logicamente distinto e una stessa area di memoria non può essere indirizzata da processori diversi.
- Lo stesso indirizzo fisico su due processori fa riferimento a due locazioni di memoria diverse.
- Solitamente si realizza fisicamente in modo distribuito
- Non c’è bisogno di protocolli software
- Non ci sono problemi per la coerenza della cache

11.3.2 Organizzazione fisica della memoria

Centralizzata

UMA Uniform Memory Access time.

Non importa quale processore richiede quale dato, il **tempo di accesso alla memoria è uniforme**.

Distribuita

NUMA Non Uniform Memory Access time.

Il **tempo di accesso non è uniforme**: dipende dalla posizione del processore e dall’indirizzo del dato richiesto.

11.3.3 Esempi di architetture

Centralized Shared Memory (CSM) o “Symmetric MultiProcessors” (SMP) È una architettura:

- Centralizzata UMA.
- Multiprocessore simmetrica (SMP).

La maggior parte dei multiprocessori in vendita sono costruiti con una architettura SMP e un basso numero di cores (di solito massimo 8). Ci sono solitamente più livelli di cache, di cui il primo privato per ogni core.

Facendo crescere il numero di core, qualunque meccanismo di controllo centrale diventa un collo di bottiglia.

Per far scalare l’architettura si può utilizzare una rete di interconnessione che porta a delle cache *shared* divise a banchi per un gruppo di core.

Small Scale MP Designs progetto multiprocessore di piccola scala.

- CSM.
 - ...e quindi **UMA**.
- Architettura a **single bus**.
- Coerenza cache tramite **snooping protocol** (vedi 11.4.1).

Distributed Shared Memory

- Pensato per numero elevato di core: da 8 a 32.
- **NUMA**: tempo non uniforme.
- Coerenza cache tramite **directory based protocol** (vedi 11.4.2).
- Le cache sono abbastanza grandi da ridurre la latenza per fare hit su un pezzo di memoria su un altro processore
- Esiste la sottoclassificazione COMA (Cache Only Memory Access).

Large scale MP Designs

- DSM
 - NUMA
 - Interconnessioni scalabili.
 - * bassa latenza
 - * alta affidabilità

11.4 Coerenza cache

Deve fornire

- Migrazione (movimento dei dati).
- Duplicazione (replicare il dato).

Come?

- Tenendo traccia dello stato di ogni blocco condiviso.

La coerenza deve garantire che:

- Tutte le letture di ogni processore devono restituire il **valore scritto più di recente**.
- Più scritture sulla stessa locazione di memoria effettuate da due processori qualsiasi devono essere viste nello stesso ordine da tutti i processori.

Due classi di protocolli:

- Snooping
- Directory based

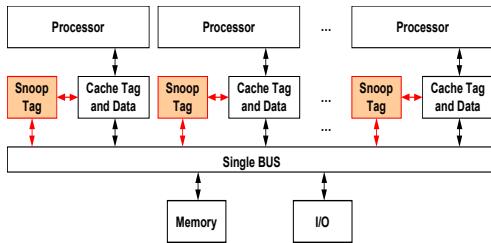
11.4.1 Snooping

Sistemi *transaction based*.

- Ogni core ha conoscenza globale della condivisione cache.
 - Tiene traccia dello stato di condivisione di ogni blocco tra le varie cache facendo *snoopying* sul bus.
- I controller monitorano il bus analizzando le richieste.
- La comunicazione avviene in broadcast.

Write invalidate

- Ogni scrittura di un blocco **invalida** le altre copie.
- È il protocollo più comune.
- Procedura ideale:
 - Ogni cache ha una copia del blocco e dello stato del blocco. Non viene centralizzato lo stato del blocco.
 - Viene mandata una richiesta per i dati condivisi in broadcast a tutti i processori
- Problema:
 - Quando mandiamo le richieste in broadcast, tutti i processori devono controllare la loro cache e questo può interferire con le loro attività, mandandoli in stall.
 - Risolviamo mettendo una porta di lettura aggiuntiva sui tag della cache del processore, di modo che la richiesta in broadcast ci può accedere direttamente senza interferire con il processore.



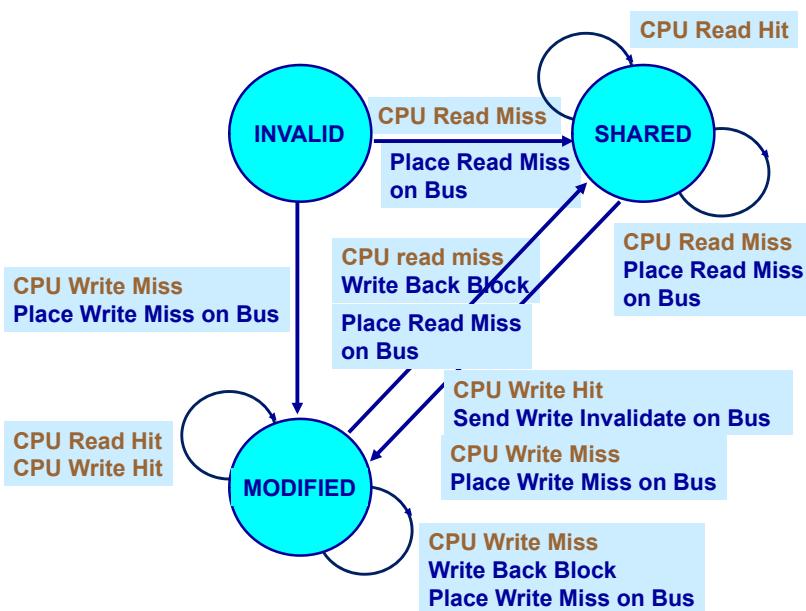
- Procedura da adottare:
 - Il processore che vuole scrivere manda in broadcast una richiesta di invalidate di tutte le altre cache, prima di scrivere. Poi scrive.
 - Le cache degli altri processori controllano se hanno quel blocco di cache e marciano invalidato.
- Questo schema permette letture multiple ma **scritture singole**.
- Questa procedura viene adottata **solo nella prima scrittura** per invalidare le altre copie! Le write dopo non mandano nulla in broadcast sul bus.
- Questo protocollo ha vantaggi simili al protocollo di *write-back* in termini di riduzione delle richieste sul bus.
- L'accesso al bus è mutualmente esclusivo, è questo a “serializzare” il processo nel punto critico.
 - L'arbitraggio del bus può diventare un problema

Ogni blocco di memoria può essere in uno dei seguenti tre stati:

- **Modified**: È stato modificato in UNA cache
- **Shared**: Aggiornato in cache e memoria
- **In nessuna cache**

Ogni blocco di cache può essere in uno dei seguenti tre stati:

- **Modified**: La cache ha un blocco dirty scrivibile, non può essere condiviso
- **Shared**: Il blocco è clean e può essere letto
- **Invalid**: Il blocco non contiene dati validi



Con la variante MESI si introduce lo stato **Exclusive**, nel quale si trova un blocco valido e aggiornato nella cache e nessuna copia di esso esiste in un'altra cache. Quando un blocco in E viene modificato non è necessario inviare notifiche di invalidazione.

Write update

- Ogni scrittura di un blocco aggiorna il valore delle altre copie.
- È più costoso in termini di prestazioni per le write.
- Meno utilizzato.
- Procedura:
 - Il processore che scrive manda in broadcast la nuova versione di un blocco.
- Abbiamo il vantaggio di ridurre la latenza perché rispetto a write-invalide, i dati tornano in cache prima.

Confronto tra write invalidate e write update

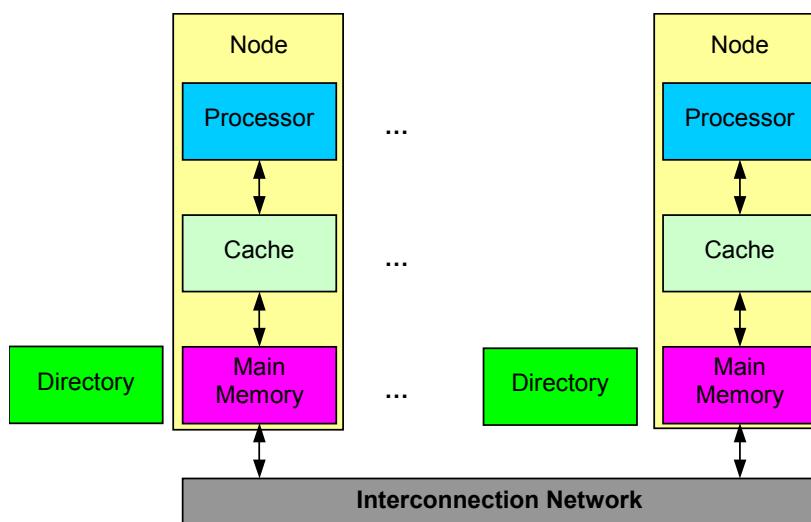
- write **invalidate**
 - richiede una transazione per ogni write
 - usa località spaziale: una transazione per blocco
- write **update**
 - genera meno latenza tra write e read.
 - aumenta il consumo di banda per ridurre la latenza.

11.4.2 Directory based

Per sistemi *message oriented* NUMA.

Lo stato condiviso di ogni blocco è mantenuto in un'**unica locazione** chiamata directory.

- Esiste un'unica directory per SMP.
- Per le DSM ci sono più directory (eventualmente distribuite).
 - Una per ogni main memory.
 - Entry distribuite e **non replicate**.
- Niente broadcast.
 - Comunicazione *point-to-point* tra i processori.
- **Scalano** meglio rispetto ai protocolli snooping.



Ogni entry contiene informazioni riguardo:

- **Stato** del blocco.
- **Quali processori hanno una copia** del blocco.

- Quale processore è il **proprietario** del blocco.

Ogni blocco nella directory può essere nello stato:

- **Uncached**: nessuno ha la copia del blocco
- **Shared**: uno o più processori hanno una copia e la memoria è aggiornata
- **Modified**: Solo un processore ha la copia aggiornata. La memoria non è aggiornata.

Ogni blocco in cache può essere nello stato:

- **Invalid**: Il blocco non contiene dati validi
- **Shared**: il blocco è valido/aggiornato ed è condiviso tra vari processori
- **Modified**: Il blocco è in accesso esclusivo e questo processo ne è il proprietario

I nodi coinvolti sono:

- Local node: dove parte la richiesta
- Home node: dov'è fisicamente la memoria richiesta
- Remote node: nodo con una copia esclusiva o shared del dato

Le operazioni fondamentali da gestire sono:

- Read miss.
- Write su blocco shared.

Tipi di messaggio

read miss da *Local* a *Home dir*. Specifica id sorgente e indirizzo. Richiede lettura.

write miss da *Local* a *Home dir*. Specifica id sorgente e indirizzo. Richiede scrittura.

data value reply da *Home dir* a *Local*. Fornisce valore.

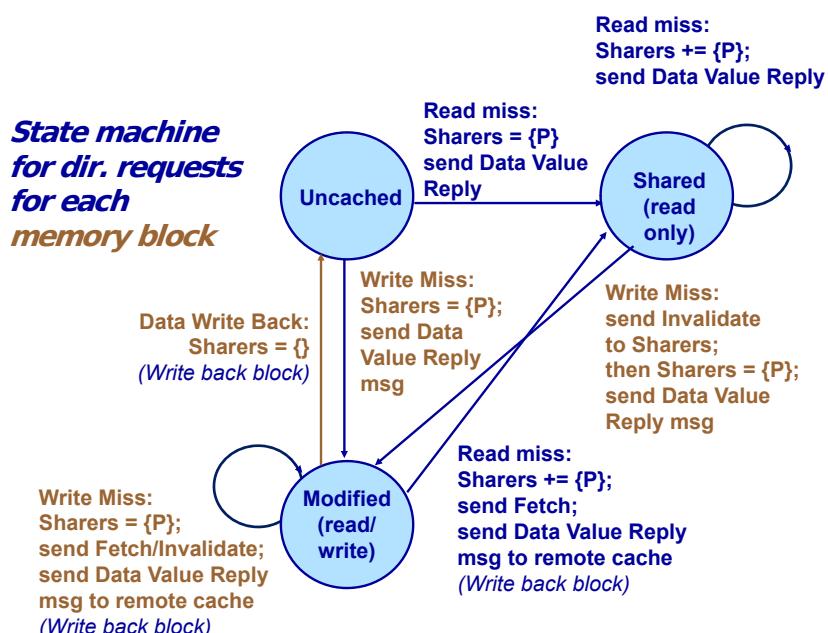
invalidate da *Local* a *Home dir* oppure da *Home dir* a *Owner*. Specifica indirizzo da invalidare.

fetch da *Home dir* a *Owner*. Specifica indirizzo. Richiede valore in condivisione.

fetch/inv da *Home dir* a *Owner*. Specifica indirizzo. Richiede di acquisire valore e diritto di proprietà.

data write back da *Owner* a *Home dir*. Specifica indirizzo e valore. Scrive un valore nella home.

11.4.3 Directory FSA



11.4.4 Modelli di comunicazione

Shared Memory

- architetture MIMD.
- Comunicazione attraverso spazio di indirizzamento condiviso.
- Conviene su piccola scala.
- Vantaggi:
 - Facile da programmare.
 - Facile da gestire (direttamente in hardware).
 - Bassa latenza.
- Ogni processore può indirizzare ogni locazione fisica, condivisa via time sharing.
- Fa uso di memoria virtuale per mappare memoria remota.

Message Passing

- Architetture MIMD.
- Ogni processore ha uno spazio di memoria privato.
- Vantaggi:
 - Meno hardware.
 - Solo le operazioni non locali sono costose
- Comunicazione I/O esplicita.
 - Send / Receive message.
 - Necessario esplicitare il buffer a cui accedere.
 - Può inviare solo ai vicini nella topologia.
- Solitamente sincrono.
 - Possibili estensioni con DMA e send non bloccanti.

Data parallel

- Architetture SIMD.
- Un processore di controllo invia comandi in broadcast a multipli elementi di esecuzione.
- Necessita di sincronizzazione globale.
 - Sono comuni i meccanismi a barriera.

Parte V

Valutazione delle prestazioni

12 Confrontare diverse esecuzioni

- Base di confronto.
- Metriche di valutazione.
 - Throughput.
 - Latenza.

12.1 Formule

Per ottenere i corrispondenti asintotici, annullare il parametro $\#cicli_{warm\ up}$.

Per ottenere i corrispondenti ideali per l'architettura, annullare i parametri $\#stalli$, $\#cicli_{warm\ up}$.

12.1.1 IPC, CPI

$$IPC = \frac{Instruction\ Count}{\#cicli_{CLK} + \#stalli + \#cicli_{warm\ up}}$$

$$CPI = \frac{1}{IPC} = \frac{\#cicli_{CLK} + \#stalli + \#cicli_{warm\ up}}{IC}$$

12.1.2 Speedup

$$Speedup = \frac{tempo\ esecuzione_B}{tempo\ esecuzione_A}$$

Chi è più veloce?

$$A \text{ è } n\% \text{ più veloce di } B \Leftrightarrow \frac{tempo\ esecuzione_B}{tempo\ esecuzione_A} = 1 + \frac{n}{100}$$

12.1.3 CPU time

$$CPU\ time = \frac{\#cicli_{CLK}}{f_{CLK}}$$

$$CPU\ time = IC \cdot CPI \cdot T_{CLK} = \frac{IC \cdot CPI}{f_{CLK}}$$

12.1.4 MIPS

Millions of Instruction Per Second

$$MIPS = \frac{IC}{tempo\ esecuzione \cdot 10^6} = \frac{f_{CLK}}{CPI \cdot 10^6}$$

12.1.5 Quali metriche utilizzare

È possibile misurare le performance a diversi livelli, dall'hardware del singolo transistor all'ambiente applicativo mese per mese. I primi sono troppo distanti dal reale impiego a cui è destinato il processore; i secondi sono troppo specifici e di complicata misurazione. Una ragionevole via di mezzo sta nel calcolare il parametro MIPS.

12.2 Legge di Amdahl

Supponendo di introdurre un miglioramento E che riduce di S il tempo di esecuzione limitatamente alla frazione di codice F .

$$tempo\ esecuzione_{con\ E} = \left((1 - F) + \frac{F}{S} \right) \cdot tempo\ esecuzione_{senza\ E}$$

$$Speedup_{con\ E} = \frac{1}{(1 - F) + \frac{F}{S}}$$

12.3 Basi di confronto

12.3.1 Microbenchmarks

- Pro:

- si vedono subito picchi di performance e colli di bottiglia.

- Contro:

- molto poco significativo.

12.3.2 Nucleo dell'applicazione

- Pro:
 - facile da eseguire anche in fasi preliminari di progettazione.
- Contro:
 - facile avere risultati fuorvianti.

12.3.3 Applicazione intera

- Pro:
 - Portable.
 - Molto usato.
 - I miglioramenti misurati sono reali.
- Contro:
 - Non completamente rappresentativo.

12.3.4 A carico effettivo

- Pro:
 - Rappresentativo della situazione.
- Contro:
 - Molto specifico.
 - Difficile da eseguire, misurare.
 - Non portabile.
 - Difficile identificare cosa funziona e cosa no.

12.4 Prestazioni della cache

12.4.1 AMAT

AMAT Average Memory Access Time. Tempo medio di accesso alla memoria. Principale metrica per misurare le performance della memoria.

$$AMAT = (Hit Rate \cdot Hit Time) + (Miss Rate \cdot Miss Time) = Hit Time + Miss Rate \cdot Miss Penalty$$

Si può poi verificare come variazioni di questo parametro influenzino il CPU time.

12.4.2 Alcune definizioni

Hit Time tempo necessario a recuperare un valore presente in cache.

Miss Time tempo necessario a recuperare un valore non in cache.

Miss Penalty = $Miss Time - Hit Time$.

Local miss miss a livello della cache considerata.

Global miss miss in tutti i livelli di cache.