


Trend di sviluppo delle pipeline

Prof. Alberto Borghese
Dipartimento di Informatica
alberto.borghese@unimi.it

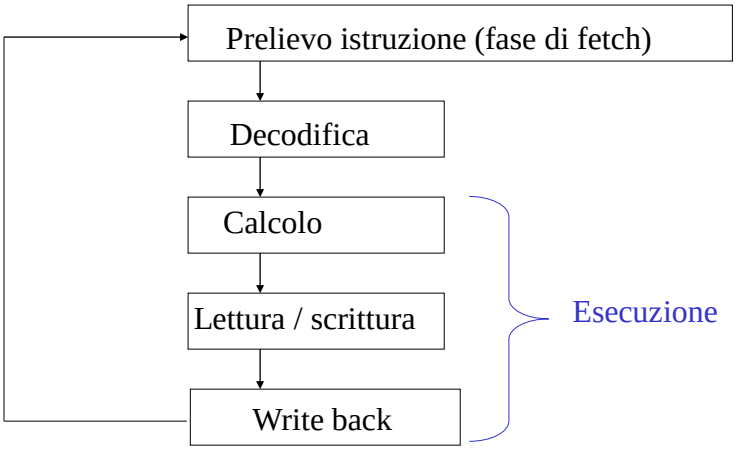
Università degli Studi di Milano

Capitoli Patterson 3.6, 3.7, 4.10, 4.11, 6.3

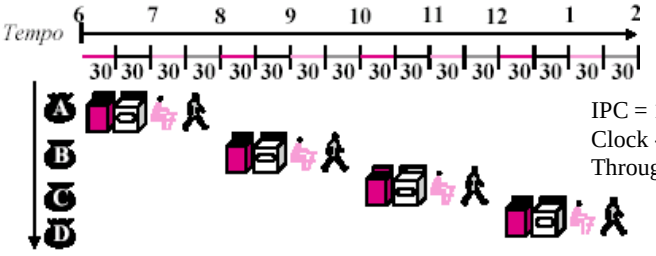

Sommario

Superpipeline
Multiple-Issue
Architetture SIMD

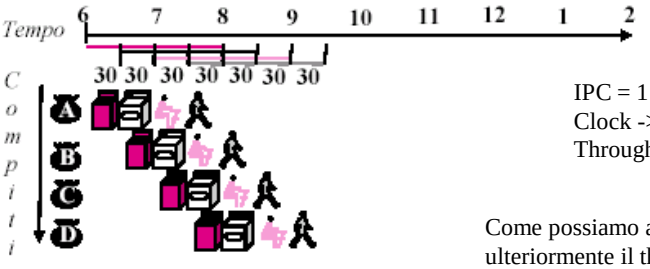
Ciclo di esecuzione di un'istruzione MIPS



Lavanderia con pipeline



IPC = 1
Clock -> 2 ore
Throughput: 0.5 bucati / h

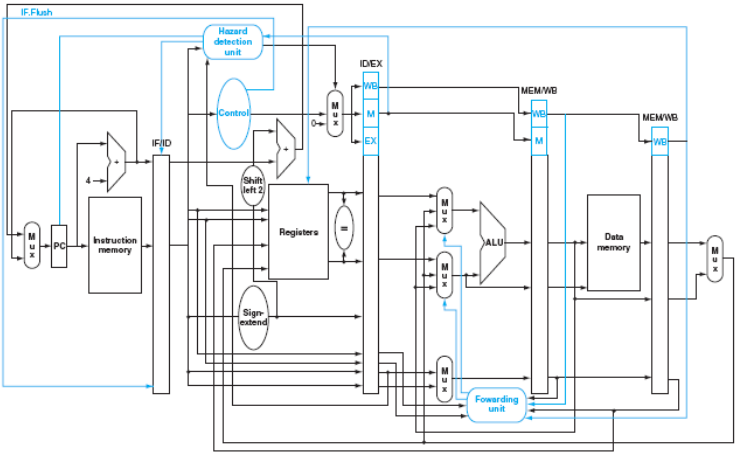


IPC = 1
Clock -> 30 minuti
Throughput: 2 bucati / h

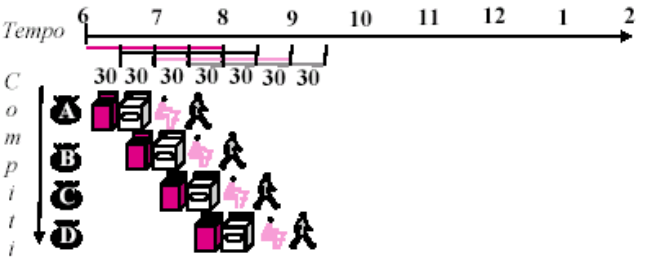
Come possiamo aumentare ulteriormente il throughput?

CPU con pipeline completa della gestione degli hazard

Throughput aumenta linearmente con il numero degli stadi (profondità)



Come creare pipeline più profonde



Occorre suddividere gli stadi di base in stadi più semplici (e.g. lavaggio -> "lavaggio" + "risciacquo; "asciugatura" = "centrifuga + asciugatura" …).

Problemi:

• Bilanciamento del carico di lavoro: occorre ottenere stadi che hanno approssimativamente lo stesso cammino critico (stessa durata). Il clock è unico.

• Criticità sui dati: stalli più frequenti.

• Criticità sul controllo: numero maggiore di stadi di cui annullare l'esecuzione (flush).

Superpipeline

Superpipeline (pipeline più profonda). Throughtput = $\mathbf{N_{stadi}} * f_{clock}$

MIPS ha 5 istruzioni in esecuzione per ogni ciclo di clock (IPC = 5).

Una pipeline con 10 stadi avrebbe un IPC = 10 teorico.

Problemi: aumento degli hazard, complessità circuitale, potenza assorbita.

Microprocessore	Anno	Frequenza di clock	Stadi di pipeline	Ampiezza del pacchetto	Esecuzione fuori ordine / speculazione	Core per chip	Potenza assorbita
Intel 486	1989	25 MHz	5	1	No	1	5W
Intel Pentium	1993	66 MHz	5	2	No	1	10W
Intel Pentium Pro	1997	200 MHz	10	3	Sì	1	29W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Sì	1	75W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Sì	1	103W
Intel Core	2006	3000 MHz	14	4	Sì	2	75W
Intel Core i7 Nehalem	2008	3600 MHz	14	4	Sì	2–4	87W
Intel Core Westmere	2010	3730 MHz	14	4	Sì	6	130W
Intel Core i7 Ivy Bridge	2012	3400 MHz	14	4	Sì	6	130W
Intel Core Broadwell	2014	3700 MHz	14	4	Sì	10	140W
Intel Core i9 Skylake	2016	3100 MHz	14	4	Sì	14	165W
Intel Ice Lake	2018	4200 MHz	14	4	Sì	16	185W

Figura 4.73 Complessità della pipeline, numero di core e potenza dissipata da alcuni microprocessori Intel. Gli stadi della pipeline del Pentium 4 non comprendono gli stadi dell'unità di consegna. Se includessimo anche questi, la profondità della pipeline del Pentium 4 sarebbe ancora maggiore.

La profondità è stabile negli ultimi 20 anni

Sommario

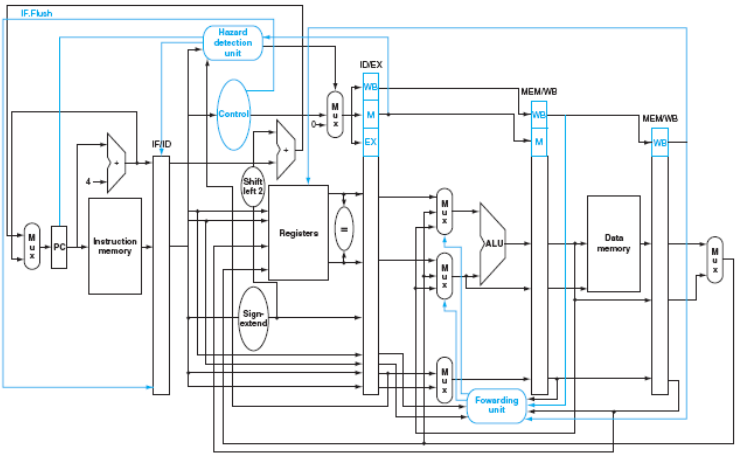
Superpipeline

Multiple-Issue

Architetture SIMD

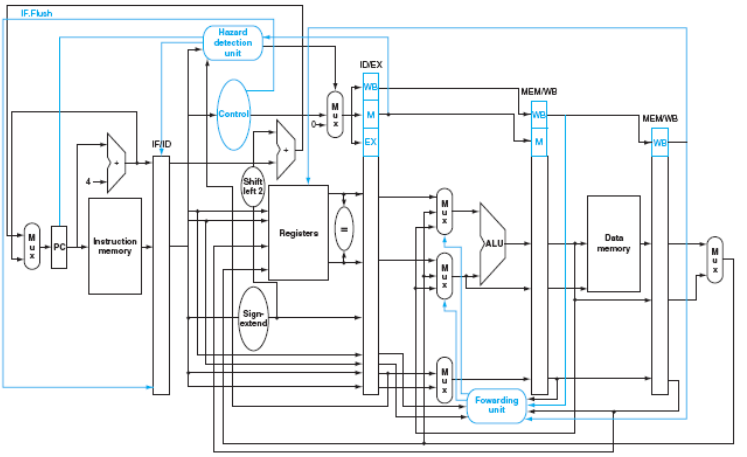
CPU con pipeline completa della gestione degli hazard

Throughput aumenta linearmente con il numero degli stadi (profondità)



CPU con pipeline completa della gestione degli hazard

Throughput aumenta linearmente con il numero degli stadi (profondità)

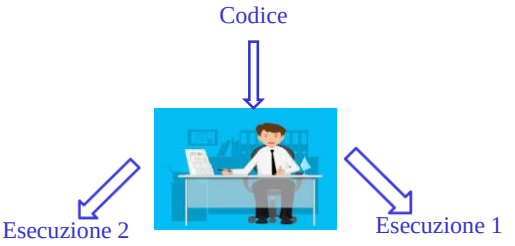


Throughput aumenta linearmente con l'ampiezza della pipeline

Pipeline avanzate: esecuzione parallela

Vengono avviate a esecuzione due istruzioni o più **simultaneamente** (moltiplicazione delle unità funzionali => ***cammini paralleli di esecuzione***)

Instruction level parallelism (ILP), parallelismo implicito (a livello dell'esecuzione delle istruzioni).



"Multiple-issue" (esecuzione parallela).

Static multiple issues (ordine delle istruzioni deciso dal compilatore)

Dynamic multiple issues (ordine delle istruzioni deciso run-time dalla CPU).

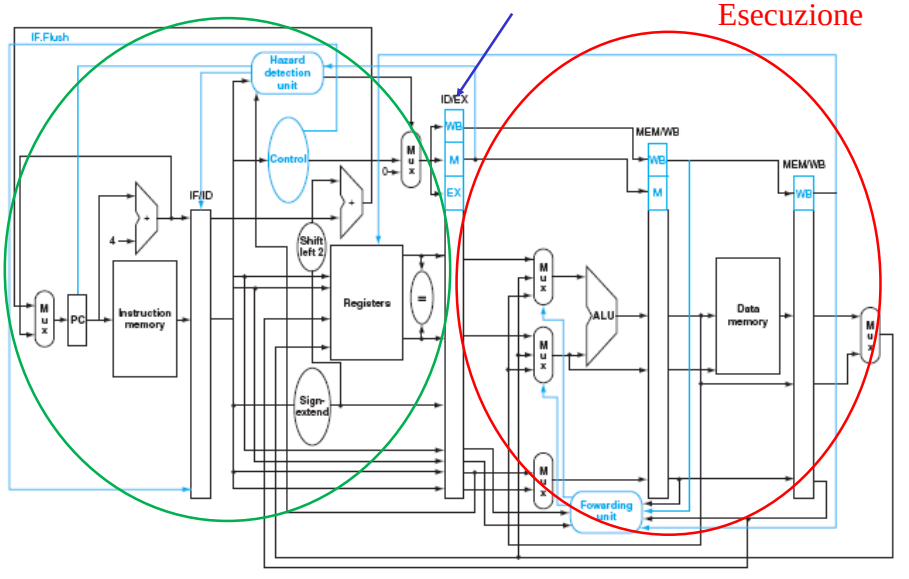
Corrisponde alla suddivisione del lavoro tra SW e HW, cioè tra il compilatore e il processore (anche nell'identificazione e soluzione degli **hazard sui dati e sul controllo**).

CPU con pipeline

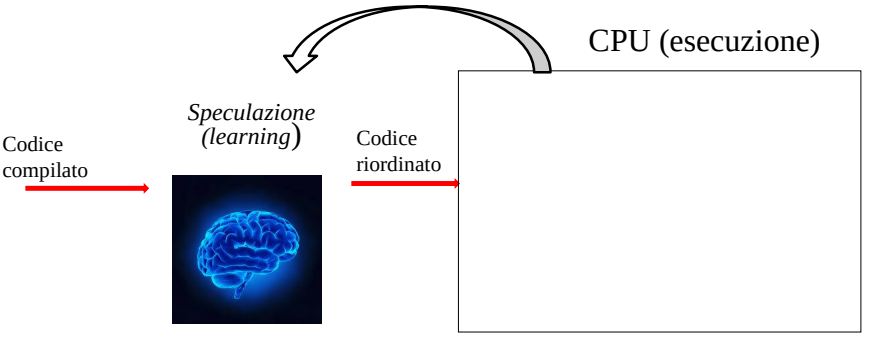
Fetch & decode

Interfaccia

Esecuzione



Fetch&Decode: CPU con speculazione



La speculazione serve per predire cosa farà il codice e per riordinarlo.

- SW (compilatore, multiple-issue statiche)
- HW (scheduler, multiple-issue dinamiche)

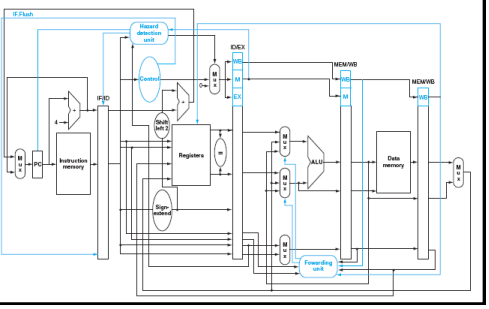
Speculazione

Predizione del risultato di una branch => scambiare codice di destinazione del salto con il codice successivo alla branch.

Predizione che l'indirizzo di una store seguita da una load non sia lo stesso (indirizzo = base_address + offset, sono molti i modi di ottenere lo stesso indirizzo) => introdurre istruzioni tra store e load per evitare stalli.

Predizione del segmento di codice a cui si salta in un salto indiretto (jal / jr)

Il compilatore e/o il processore provvedono a riordinare l'ordine di trasmissione delle istruzioni alla CPU (parallelismo statico) o agli stadi della CPU che provvedono all'esecuzione del codice.



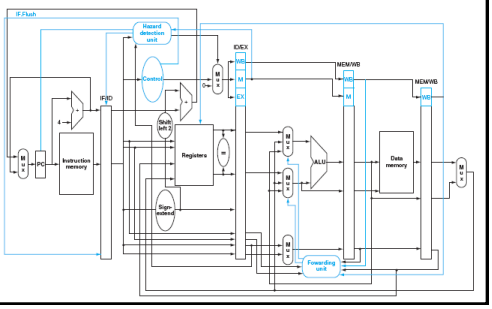
Speculazione errata

Se la speculazione non è corretta => **roll-back:** Eliminare ogni effetto delle istruzioni eseguite “per sbaglio” (cf. flush pipeline quando la predizione di una branch è risultata sbagliata) e ripristino dello stato della CPU.

Multiple-issue statiche. Vengono introdotte ulteriori istruzioni dal compilatore, che nel caso in cui la speculazione non risulta corretta, rimandano a una **procedura di fix-up** (salto codizionato, esecuzione condizionata)**.**

Multiple-issue dinamiche. La CPU **mantiene i risultati in un buffer** (cf. MEM/WB) **fino a quando la speculazione non è stata risolta**: il risultato viene scritto nel register file o in memoria oppure viene semplicemente **scartato** se la speculazione risulta errata.

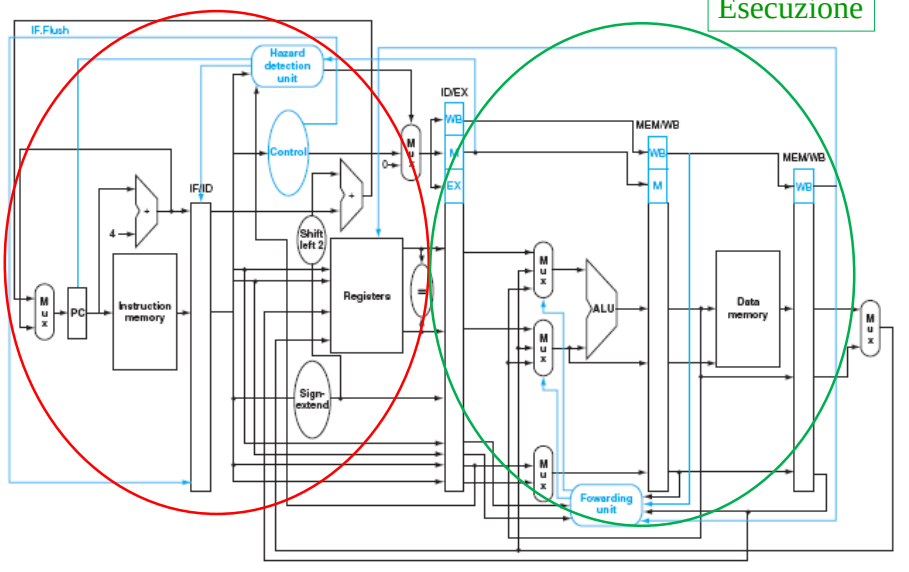
Cosa fare delle eccezioni che vengono sollevate su un’istruzione che viene eseguita in base a una speculazione errata?



Multiple-issue statica

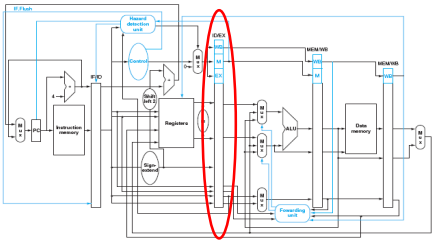
Fetch & decode

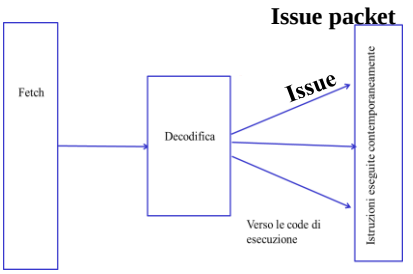
Esecuzione



Issue packet

Multiple issue = a esecuzione parallela



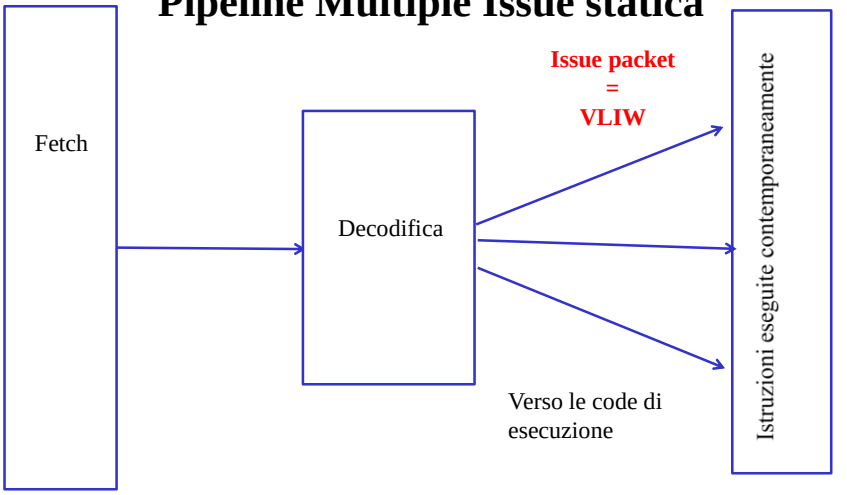


Issue packet (informazioni avviate sul data-path dopo la decodifica):

- Dati
- Registri
- Tipo operazione
- **Può comprendere più istruzioni avviate in parallel-**

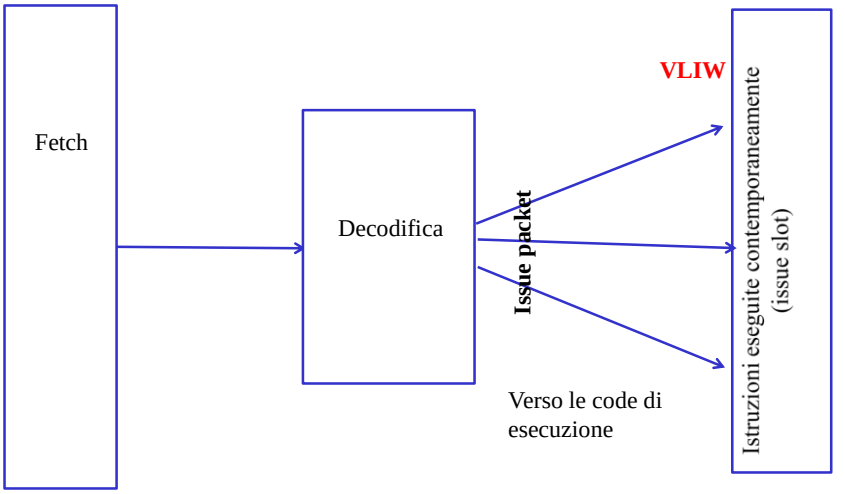
Gli issue vengono generati dagli stadi Fetch&Decode e dipendono dalla struttura delle fasi di esecuzione della CPU

Fetch&Decode
Pipeline Multiple Issue statica



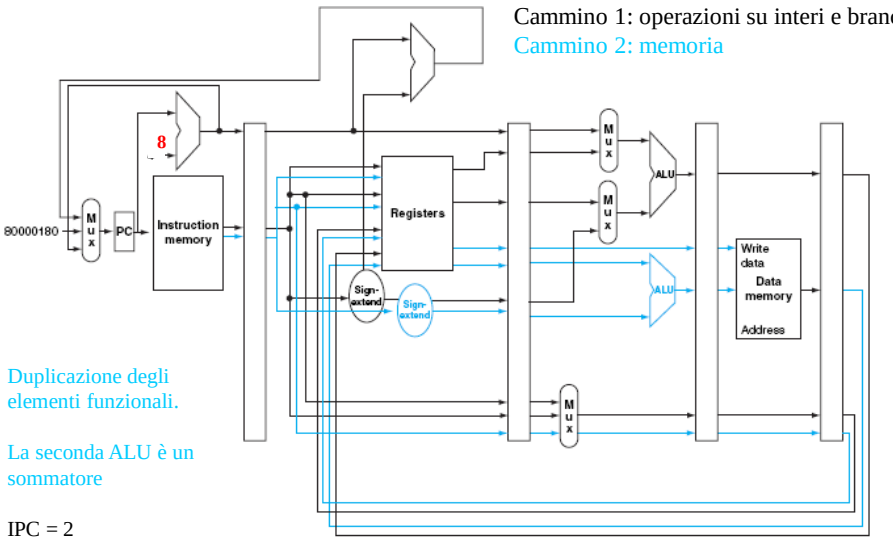
Riorganizzazione del codice da parte del compilatore

Struttura Pipeline Multiple Issue statica



Issue packet can be viewed as a Very-Long-Instruction-Word (VLIW)
Esistono dei vincoli su quali istruzioni inserire nello stesso issue packet
L'ordine di esecuzione viene deciso dal compilatore.
Primi progetti: Itanium (2000), Itanium-2 (2002) by Intel.

Multiple-issue statica: il MIPS64



Cammino 1: operazioni su interi e branch
Cammino 2: memoria

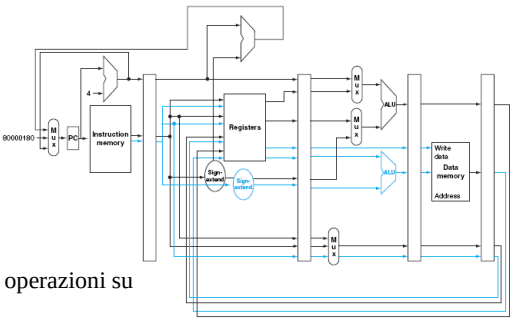
Duplicazione degli elementi funzionali.

La seconda ALU è un sommatore

IPC = 2

Multiple-issue statica: il MIPS64

Tipo di istruzioni	Stadi di pipeline							
ALU o salto	IF	ID	EX	MEM	WB			
Load o store	IF	ID	EX	MEM	WB			
ALU o salto		IF	ID	EX	MEM	WB		
Load o store		IF	ID	EX	MEM	WB		
ALU o salto			IF	ID	EX	MEM	WB	
Load o store			IF	ID	EX	MEM	WB	
ALU o salto				IF	ID	EX	MEM	WB
Load o store				IF	ID	EX	MEM	WB



VLIW di 64 bit.

Fetch e decodifica semplicata:

Istruzione 1 => Cammino 1: operazioni su interi e branch

Istruzione 2 => Cammino 2: operazioni su memoria

Problemi nella creazione dello issue packet

Hazard sul controllo. Le due istruzioni nei branch delay slot possono dovere essere eliminate dalla pipeline (nel caso in cui la predizione sul salto si riveli errata).

Hazard sui dati oltre a quelli insiti nel codice sequenziale, si generano **hazard dovuti alla parallelizzazione del codice**.

add $t0, $t1, $t2
lw $s0, 40($t0)

Non provoca stallo su una pipeline single issue ma provoca stallo di 1 istruzione nella pipeline a due vie:

	Q1	Q2
è errata:	~~add $t0, $t1, $t2~~	~~lw $s0, 40($t0)~~
è corretta:	add $t0, $t1, $t2	nop
	nop	lw $s0, 40($t0)

Hazard sui dati (stall on load).

lw $s0, 40($t0)
nop
add $t0, $t1, $s0

Provoca stallo di 1 istruzione (1 slot) su una pipeline single issue

Provoca stallo di 2 istruzioni (2 slot) nella pipeline a due vie:

	Q1	Q2
è errata:	~~add $t0, $t1, $s0~~	~~lw $s0, 40($t0)~~
è corretta:	nop	lw $s0, 40($t0)
	nop	nop
	add $t0, $t1, $s0	nop

Esempio di costruzione dei pacchetti

```
Ciclo: lw   $t0, 0($s1)     #vett[i] -> $t0   ($s1 ind di vett[N])
       addu $t0, $t0, $s2   #vett[i] + $s2
       sw   $t0, 0($s1)     #vett[i] <- $t0
       addi $s1, $s1, -4    #vett[i-1]
       bne  $s1, $s0, Ciclo  # until vett[0] (&vett[0] = $s0)
       or   $s6, $s7, $s5
```

A ogni iterazione leggiamo vett[i], sommiamo s2, scriviamo vett[i] =>

```
for (i=N; i=1; i--) vett[i] = vett[i] + s2;
```

Vincoli per il compilatore (codice) in distanza tra istruzioni:

- ***addi*** non può essere eseguita prima della ***lw*** (1 stadio) perchè l'indirizzo deve essere decrementato successivamente alla lettura dalla memoria.
- ***addi*** non può essere eseguita prima della ***sw*** (1 stadio) per lo stesso motivo.
- ***addu*** ha bisogno del contenuto di $t0 all'inizio della fase EXE, questo viene disponibile solo nella fase di WB della ***lw*** (2 stadi).
- ***sw*** ha bisogno del contenuto di $t0 all'inizio della fase di MEM, questo viene disponibile solo nella fase di MEM della ***addu*** (1 stadio)
- ***bne*** ha bisogno del contenuto di $s1 all'inizio della fase DEC, questo viene disponibile solo nella fase di MEM della ***addi*** (2 stadi)

Vincoli per il compilatore (struttura):

- 2 cammini di esecuzione: Q1 per interi e salti condizionati, Q2 per accesso alla memoria.

Esempio di costruzione dei pacchetti

```
Ciclo: lw   $t0, 0($s1)     #vett[i] -> $t0
       addu $t0, $t0, $s2   #vett[i] + $s2
       sw   $t0, 0($s1)     #vett[i] <- $t0
       addi $s1, $s1, -4    #vett[i-1]
       bne  $s1, $s0, Ciclo
       or   $s6, $s7, $s5
```

Ciclo:	nop	lw $t0, 0($s1)	1
	nop	nop	2
	addu $t0, $t0, $s2	nop	3
	addi $s1, $s1, -4	sw $t0, 0($s1)	4
	nop	nop	5
	bne $s1, $s0, Ciclo	nop	6

IPC = 0,8 < 1 (6 cicli di clock x 5 istruzioni)….
Non sfrutto la parallelizzazione
Si può fare meglio?

Vincoli per il compilatore (codice):

- ***addi*** non può essere eseguita prima della ***lw*** (1 stadio).
- ***addi*** non può essere eseguita prima della ***sw*** (1 stadio).
- ***addu*** ha bisogno del contenuto di $t0 all'inizio della fase EXE, questo viene disponibile solo nella fase di WB della ***lw*** (2 stadi).
- ***sw*** ha bisogno del contenuto di $t0 all'inizio della fase di MEM, questo viene disponibile solo nella fase di MEM della ***addu*** (1 stadio)
- ***bne*** ha bisogno del contenuto di $s1 all'inizio della fase DEC, questo viene disponibile solo nella fase di MEM della ***addi*** (2 stadi)

Vincoli per il compilatore (struttura):

- 2 cammini di esecuzione: Q1 per interi e salti condizionati, Q2 per accesso alla memoria.

Riordinamento del codice

```
Ciclo: lw    $t0,  0($s1)
       addi  $s1,  $s1,  -4
       addu  $t0,  $t0,  $s2
       sw    $t0,  4($s1)
       bne   $s1,  $s0,  Ciclo
       or    $s6,  $s7,  $s5
```

Da 6 cicli di clock a 4 cicli di clock per ogni iterazione. IPC = 0,8 -> IPC = 1.2!!!
Lontano da IPC = 2

NB $s1 viene decrementato di 4 prima

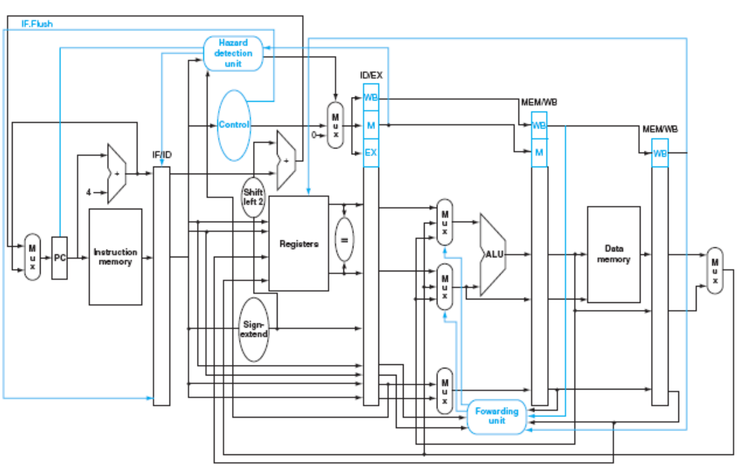
Ciclo:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1, -4	nop	2
	addu $t0, $t0, $s2	sw $t0, 4($s1)	3
	bne $s1, $s0, Ciclo	nop	4
	or $s6, $s7, $s5	nop	

Vincoli per il compilatore (codice):

- ***addi*** non può essere eseguita prima della ***lw*** (1 stadio).
- ***addi*** non può essere eseguita prima della ***sw*** (1 stadio).
- ***addu*** ha bisogno del contenuto di $t0 all'inizio della fase EXE, questo viene disponibile solo nella fase di WB della ***lw*** (2 stadi).
- ***bne*** ha bisogno del contenuto di $s1 all'inizio della fase DEC, questo viene disponibile solo nella fase di MEM della ***addi*** (2 stadi)
- ***sw*** ha bisogno del contenuto di $t0 all'inizio della fase di MEM, questo viene disponibile solo nella fase di MEM della ***addu*** (1 stadio)

Vincoli per il compilatore (struttura):

CPU con pipeline a singolo issue: miglioramento dei cicli



Analisi del ciclo (CPU a singolo issue)

```
Ciclo:  lw   $t0, 0($s1)        # M[s1] -> t0
        addu $t0, $t0, $s2      # t0 = t0 + s2
        sw   $t0, 0($s1)        # M[s1] <- t0
        addi $s1, $s1, -4       # next element
        bne  $s1, $s0, Ciclo
        or   $s6, $s7, $s5
```

5 istruzioni per ogni ciclo:
- 3 corpo del ciclo
- 2 controllo

2 stalli

7 cicli di clock per ogni iterazione

```
Ciclo:  lw   $t0, 0($s1)
        nop
        addu $t0, $t0, $s2
        sw   $t0, 0($s1)
        addi $s1, $s1, -4
        nop
        bne  $s1, $s0, Ciclo

        or   $s6, $s7, $s5
```

Durata di 4 iterazioni del ciclo = 7 * 4 = 28 cicli di clock

Possiamo ridurre la durata a 14 cicli di clock! (e a 8 cicli di clock su una CPU con pacchetti di 2 issue).

Srotolamento del ciclo 4 iterazioni)

```
Ciclo:  lw   $t0, 0($s1)        # M[s1] -> t0
        addu $t0, $t0, $s2      # t0 = t0 + s2
        sw   $t0, 0($s1)        # M[s1] <- t0
        lw   $t0, -4($s1)       # M[s1-4] -> t0
        addu $t0, $t0, $s2      # t0 = t0 + s2
        sw   $t0, -4($s1)       # M[s1-4] <- t0
        lw   $t0, -8($s1)       # M[s1-8] -> t0
        addu $t0, $t0, $s2      # t0 = t0 + s2
        sw   $t0, -8($s1)       # M[s1-8] <- t0
        lw   $t0, -12($s1)      # M[s1-12] -> t0
        addu $t0, $t0, $s2      # t0 = t0 + s2
        sw   $t0, -12($s1)      # M[s1-12] <- t0
        addi $s1, $s1, -16
        bne  $s1, $s0, Ciclo
```

```
Ciclo:  lw   $t0, 0($s1)
        addu $t0, $t0, $s2
        sw   $t0, 0($s1)
        addi $s1, $s1, -4
        bne  $s1, $zero, Ciclo

        or   $s6, $s7, $s5
```

Situazione attuale: 5 * 4 = 20 istruzioni senza tenere conto degli hazard e degli stalli.

Per evitare gli stalli devo mettere:
- Tra lw e addu almeno un'istruzione.
- Tra addi e bne almeno un'istruzione.

Ma non posso spostare la addu dopo la sw.

Srotolamento del ciclo (nop)

```
Ciclo:  lw   $t0, 0($s1)          # M[s1] -> t0
        addu $t0, $t0, $s2        # t0 = t0 + s2
        sw   $t0, 0($s1)          # M[s1] <- t0
        addi $s1, $s1, -4         # next element
        bne  $s1, $zero, Ciclo

Ciclo:  lw   $t0, 0($s1)          # M[s1] -> t0
        nop
        addu $t0, $t0, $s2        # t0 = t0 + s2
        sw   $t0, 0($s1)          # M[s1] <- t0
        lw   $t0, -4($s1)         # M[s1-4] -> t0
        nop
        addu $t0, $t0, $s2        # t0 = t0 + s2
        sw   $t0, -4($s1)         # M[s1-4] <- t0
        lw   $t0, -8($s1)         # M[s1-8] -> t0
        nop
        addu $t0, $t0, $s2        # t0 = t0 + s2
        sw   $t0, -8($s1)         # M[s1-8] <- t0
        lw   $t0, -12($s1)        # M[s1-12] -> t0
        nop
        addu $t0, $t0, $s2        # t0 = t0 + s2
        sw   $t0, -12($s1)        # M[s1-12] <- t0
        addi $s1, $s1, -16        # skip 4 elements
        nop
        bne  $s1, $s0, Ciclo
```

Partiamo da:
5 istruzioni -> 7 cicli (con le nop)
* 4 iterazioni = 28 cicli clock

Dopo lo srotolamento del ciclo:
(3 istruzioni + 1 nop) * 4 iterazioni +
2 istruzioni di controllo +
1 nop =

19 cicli di clock

Si può fare meglio?

Ridenominazione dei registri

```
Ciclo:  lw   $t0, 0($s1)
        nop
        addu $t0, $t0, $s2
        sw   $t0, 0($s1)
        lw   $t0, -4($s1)
        nop
        addu $t0, $t0, $s2
        sw   $t0, -4($s1)
        lw   $t0, -8($s1)
        nop
        addu $t0, $t0, $s2
        sw   $t0, -8($s1)
        lw   $t0, -12($s1)
        nop
        addu $t0, $t0, $s2
        sw   $t0, -12($s1)
        addi $s1, $s1, -16
        nop
        bne  $s1, $s0, Ciclo
```

```
Ciclo:  lw   $t0, 0($s1)
        nop
        addu $t0, $t0, $s2
        sw   $t0, 0($s1)
        lw   $t1, -4($s1)
        nop
        addu $t1, $t1, $s2
        sw   $t1, -4($s1)
        lw   $t2, -8($s1)
        nop
        addu $t2, $t2, $s2
        sw   $t2, -8($s1)
        lw   $t3, -12($s1)
        nop
        addu $t3, $t3, $s2
        sw   $t3, -16($s1)
        addi $s1, $s1, -16
        nop
        bne  $s1, $s0, Ciclo
```

Le operazioni all'interno di un ciclo non dipendono da quelle eseguite all'interno di altri cicli.
Introduco altri registri $t (se si può!). Ogni iterazione di ciclo ha un registro $t dedicato.

Sempre 19 cicli di clock. E quindi?

Riorganizzazione del ciclo - I

```
Ciclo:  lw   $t0, 0($s1)          # M[s1] -> t0
        lw   $t1, -4($s1)         # M[s1-4] -> t1
        lw   $t2, -8($s1)         # M[s1-8] -> t2
        lw   $t3, -12($s1)        # M[s1-12] -> t0
        addu $t0, $t0, $s2        # t0 = t0 + s2
        addu $t1, $t1, $s2        # t1 = t1 + s2
        addu $t2, $t2, $s2        # t2 = t2 + s2
        addu $t3, $t3, $s2        # t3 = t3 + s2
        sw   $t0, 0($s1)          # M[s1] <- t0
        sw   $t1, -4($s1)         # M[s1-4] <- t1
        sw   $t2, -8($s1)         # M[s1-8] <- t2
        sw   $t3, -12($s1)        # M[s1-12] <- t3
        addi $s1, $s1, -16
        nop
        bne  $s1, $s0, Ciclo

        or   $s6, $s7, $s5
```

Sempre 14 istruzioni, ma grazie alla ridenominazione dei registri, rimane solo la dipendenza sulla `bne`, 15 cicli di clock.

Riorganizzazione del ciclo - II

```
Ciclo:  addi $s1, $s1, -16
        lw   $t0, 16($s1)
        lw   $t1, 12($s1)
        lw   $t2, 8($s1)
        lw   $t3, 4($s1)
        addu $t0, $t0, $s2
        addu $t1, $t1, $s2
        addu $t2, $t2, $s2
        addu $t3, $t3, $s2
        sw   $t0, 16($s1)
        sw   $t1, 12($s1)
        sw   $t2, 8($s1)
        sw   $t3, 4($s1)
        bne  $s1, $s0o, Ciclo

        or   $s6, $s7, $s5
```

Da 4 * 7 = 28 cicli di clock a 14 cicli di clock
Risolte tutte le criticità
Raddoppio la velocità

14 istruzioni su implementazione a singolo cammino di esecuzione, **nessuna nop**

Esecuzione fuori ordine (metà tempo sulla stessa CPU a cammino di esecuzione singolo)

```
Ciclo:  lw   $t0, 0($s1)
        addu $t0, $t0, $s2
        sw   $t0, 0($s1)
        addi $s1, $s1, -4
        bne  $s1, $s0, Ciclo

        or   $s6, $s7, $s5
```

Implementazione parallelizzata

```
Ciclo:  lw   $t0, 0($s1)          # M[s1] -> t0
        addu $t0, $t0, $t2        # t0 = t0 + t2
        sw   $t0, 4($s1)          # M[s1+4] <- t0
        addi $s1, $s1, -4         # parola precedente in memoria
        bne  $s1, $zero, Ciclo
        or   $s6, $s7, $s5
```

```
Ciclo:  addi $s1, $s1, -16
        lw   $t0, 16($s1)
        lw   $t1, 12($s1)
        lw   $t2, 8($s1)
        lw   $t3, 4($s1)
        addu $t0, $t0, $s2
        addu $t1, $t1, $s2
        addu $t2, $t2, $s2
        addu $t3, $t3, $s2
        sw   $t0, 16($s1)
        sw   $t1, 12($s1)
        sw   $t2, 8($s1)
        sw   $t3, 4($s1)
        bne  $s1, $s0, Ciclo
        or   $s6, $s7, $s5
```

Ciclo:	addi $s1, $s1, -16	
	nop	lw $t0, 16($s1)
	nop	lw $t1, 12($s1)
	addu $t0, $t0, $s2	lw $t2, 8($s1)
	addu $t1, $t1, $s2	lw $t3, 4($s1)
	addu $t2, $t2, $s2	sw $t0, 16($s1)
	addu $t3, $t3, $s2	sw $t1, 12($s1)
	nop	sw $t2, 8($s1)

Da 4*7 = 28 cicli di clock a 14 cicli di clock a 9 cicli di clock

Codice più lungo, 4 registri $t, invece di 1: $t0.
Scrivere cicli semplici e modulari!!

Codice riordinato per multiple-issue a 2 vie

```
Ciclo:  addi $s1, $s1, -16
        lw   $t0, 16($s1)
        nop
        lw   $t1, 12($s1)
        addu $t0, $t0, $s2
        lw   $t2, 8($s1)
        addu $t1, $t1, $s2
        lw   $t3, 4($s1)
        addu $t2, $t2, $s2
        sw   $t0, 16($s1)
        addu $t3, $t3, $s2
        sw   $t1, 12($s1)
        nop
        sw   $t2, 8($s1)
        bne  $s1, $s0, Ciclo
        sw   $t3, 4($s1)

        or   $s6, $s7, $s5
```

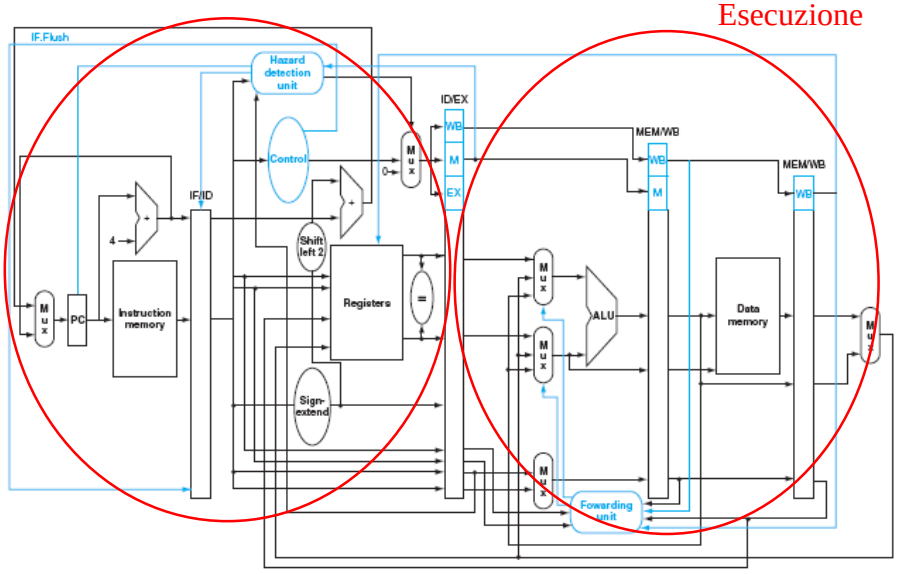
```
Ciclo:  lw   $t0, 0($s1)
        nop
        addu $t0, $t0, $s2
        sw   $t0, 0($s1)
        addi $s1, $s1, -4
        nop
        bne  $s1, $s0, Ciclo

        or   $s6, $s7, $s5
```

Modifica per multiple-issue dinamiche

Fetch & decode

Esecuzione



Dynamic multiple issues

Questi processori sono detti anche **superscalari**.

La scelta di quali istruzioni inviare alla pipe-line viene eseguita durante l'esecuzione stessa. Dipende dalla compatibilità tra le varie istruzioni e da eventuali hazard su dati e controllo.

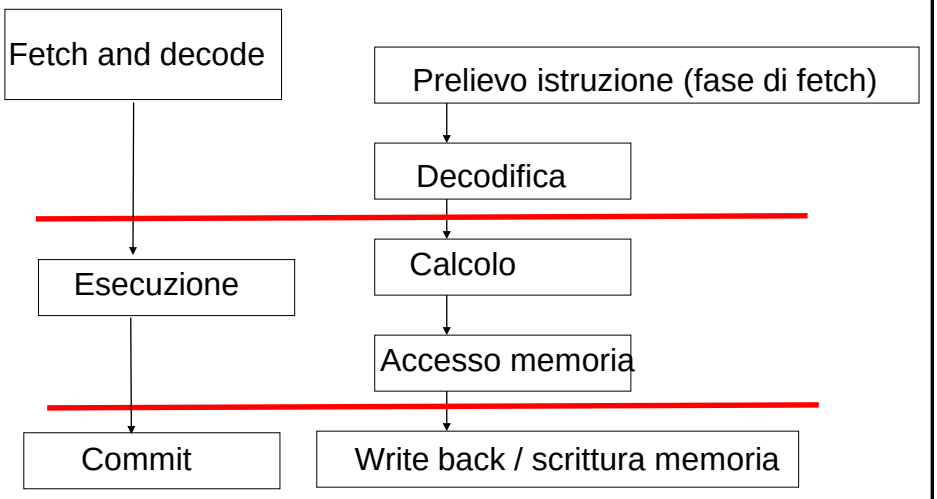
Nella versione più semplice, le istruzioni sono processate in sequenza ed il processore decide se elaborarne nessuna (stallo), una o più di una a seconda delle criticità riscontrate.

L'ottimizzazione del codice da parte del compilatore è comunque richiesta.

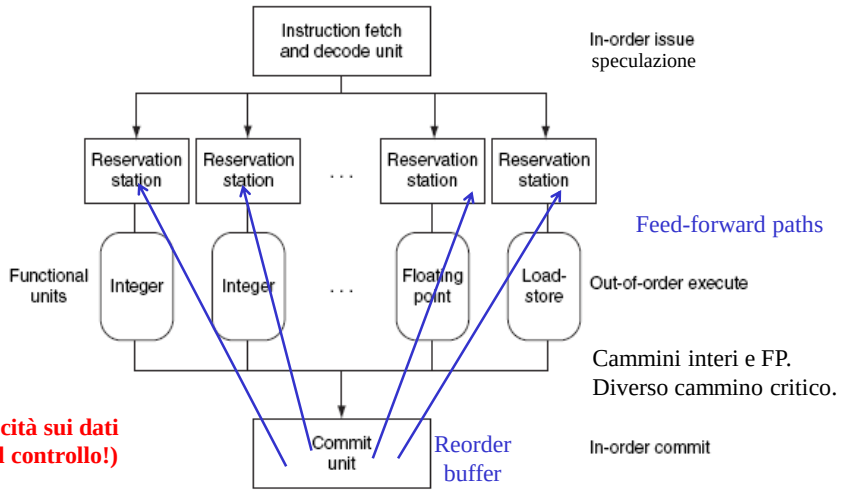
E' la CPU che produce gli issue packet inviati ad esecuzione.

La correttezza dell'esecuzione deve comunque essere garantita dalla CPU.

Ciclo di esecuzione di un'istruzione MIPS

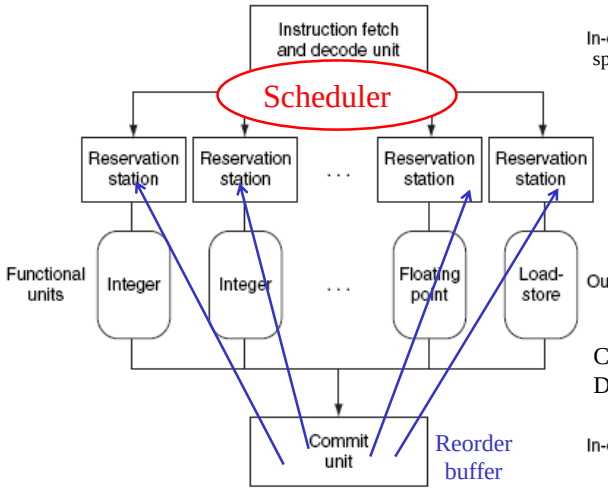


Pipeline con schedulazione dinamica



Esistono diversi cammini paralleli (per la fase di esecuzione e memoria) dell'istruzione, vengono scelti dinamicamente, run-time.

Lo scheduler



Scheduler:
- Sceglie le istruzioni
- Avvia alle diverse reservation station
- Forma dinamicamente gli issue packet

Esempio

```
add  $t0, $t1, $t2
sub  $s0, $s1, $s2
beq  $s3, $s4, salta
or $f5, $f6, $f7
lw $t4, 20($t5)
```

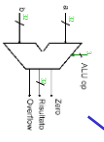
Fetch and decode (**scheduling**)

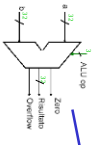
#s1, #s2, 'sub'
#s0
#t1, #t2, 'add'
#t0

#s3, #s4, 'sub'
Branch, PC, costante

#f5, #f6, 'or'
#f7

#t5, 20, 'read'
#t4





ALUfp

Memoria

Commit

Buffer da cui si trasferisce a memoria o register file

Principi della schedulazione dinamica

Obbiettivo: mettere in esecuzione istruzioni che non presentino criticità.

Le istruzioni vengono bufferizzate dalla **reservation station**, la quale gestisce la coda delle istruzioni che hanno bisogno della stessa unità funzionale.

Al termine dell'esecuzione la **commit unit**, provvede a riordinare i risultati delle istruzioni nella sequenza con la quale devono essere restituiti (**out-of-order execution, in-order commit**).

Per eseguire un'operazione è sufficiente che il dato sia già pronto nella reservation station e nel reorder buffer, contenuto nella commit unit, senza che sia necessariamente scritto nel register file.

Un'operazione viene lanciata, quando i dati sono pronti. Se un dato non è pronto viene inserita un'etichetta che associa (traccia) il dato al cammino che lo deve produrre. Quando il dato viene eseguito, tramite etichetta si libera il blocco all'esecuzione dell'istruzione.

NB Le istruzioni non sono eseguite sequenzialmente.

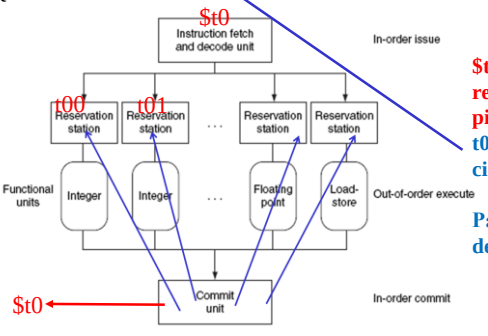
Register renaming in una superscalare

```
Ciclo: lw   $t0, 0($s1)
       addu $t0, $t0, $t2
       sw   $t0, 0($s1)
       addi $s1, $s1, -4
       bne  $s1, $zero, Ciclo
       or   $s6, $s7, $s5
```

	Istruzioni ALU o di salto condizionato	Istruzioni trasferimento dati	Ciclo di clock
Ciclo:	addi $s1, $s1, -16	lw $t0, 16($s1)	1
		lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t3, $s2	sw $t1, 12($s1)	6
		sw $t2, 8($s1)	7
	bne $s1,$s0 , Ciclo	sw $t3, 4($s1)	8

Ogni ciclo **non va** a modificare variabili che servono al ciclo successivo anche se si utilizza a ogni ciclo sempre il registro $t0. Quale sarà il valore finale di **$t0?**

Ridenominazione dei registri ($ti -> $t0i) per eliminare le **false dipendenze (false sharing)** o **antidipendenze.**



$t0 -> {t00, t01, t02, t03}, registri interni della pipeline.
t03 -> $t0 al termine del ciclo (4 iterazioni).

Parallelizzo l'esecuzione del ciclo

Le celle di memoria lette diventeranno: M[s1+16, M[s1+12], M[s1+8], M[s1+4] =>
$t0 avrà il valore di M[4]. **Nel register file viene scritto solo il valore finale.**

Roll-back

La speculazione può essere fatta sia dal compilatore che dal processore (mediante logica di controllo).

E se la speculazione risulta sbagliata? Deve esistere un meccanismo di correzione (**roll-back**). La speculazione si paga in termini di meccanismi per **controllare** se la speculazione è stata corretta e di **correggerla**.

Nelle **multiple-issue statiche**, il **compilatore** inserisce delle istruzioni di controllo e di correzione a speculazioni errate, anche chiamando procedure opportune che correggono quanto fatto (e.g. Procedure che eseguono le operazioni inverse erroneamente eseguite).

Nell **multiple-issue dinamiche**, il **reorder buffer** colleziona i risultati, che vengono scritti nel register file solamente quando la speculazione è stata verificata come corretta. Ciascun registro del riorder buffer contiene il mapping ai registri interni di pipeline. **Il flush avviene cancellando la mappatura (invalidando il contenuto del buffer associato).**

Occorre speculare quando si hanno degli elementi validi, altrimenti si possono inserire problemi (vedi eccezioni generate dall'esecuzione di un'istruzione sbagliata o con dati sbagliati, eccezioni "speculative") che rendono il funzionamento meno efficiente.

Codice più semplice (anche se più lungo) e modulare è più efficiente!

Renaming e roll-back

- La CPU mette in buffer i risultati dell'esecuzione fino a quando non si è potuto verificare la correttezza della speculazione (esecuzione condizionata).
- Nel caso di speculazione errata, la cancellazione del lavoro fatto viene ottenuta svuotando (invalidando) i buffer e correggendo la sequenza di istruzioni (meccanismo di **roll-back**).
- Nel caso in cui l'esecuzione sia corretta, il risultato viene copiato in memoria dati e/o nel register file. Nell'esempio precedente: $t0 = $t03, viene copiato il valore più recente di $t0.
- Il register renaming può essere utilizzato anche per la gestione degli hazard => invece di correggere il register file è sufficiente cancellare l'associazione registro interno – registro del register file.
- Ampliamento del numero dei registri. Limitazione dello spilling dei registri con il renaming.

Confronto tra CPU Multiple-issue statiche e Multiple-issue dinamiche

L'hardware di una pipe-line superscalare garantisce la correttezza del codice. Il codice verrà eseguito correttamente qualunque sia la CPU sul quale viene fatto girare (purchè contenga l'ISA su cui il codice è basato!).

Nelle **multiple-issue statiche**, spesso occorre ricompilare passando da una CPU ad un'altra per evitare che il codice venga eseguito con prestazioni molto scadenti (implementazioni diverse delle pipeline, numero di registry interni diversi, profondità diversa, numero di cammini di esecuzione diverso…). Nelle multiple-issue dinamiche, ciò non è necessario.

La speculazione può essere fatta sia dal compilatore (multiple-issue statici, SW) che dal processore (multiple-issue dinamici, HW).

I meccanismi associate: riordinamento del codice, srotolamento dei cicli, register renaming può essere effettuato dalla CPU o dal compilatore.

Parallelismo statico e dinamico collaborano nel rendere veloce l'esecuzione.

Sommario

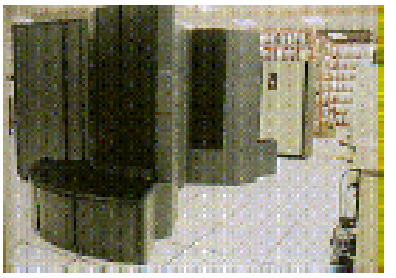
Superpipeline

Multiple-Issue

Architetture SIMD

La quarta generazione (1971-1977)

Cray I (1976) - Primo supercalcolatore. Vettoriale (SIMD)



Calcolo vettoriale

```
for (i=0;i<64;i++)            // 64 add instructions in a single loop
       C[i] = A[i] + B[i];
```

Creiamo una gerarchia di calcolo

```
for (i=0;i<16;i++)            // 16 cicli di somma di gruppi di 4
       for (j=0; j<4; j++)          // elementi
              C[i*4+j] = A[i*4+j] + B[i*4+j];

Inner cycle can be implemented in HW:

for (i=0;i<16;i++)
  C[i*4:i*4+3] = A[i*4:i*4+3] + B[i*4:i*4+3]; // singola operazione
                                                // vettoriale
```

Architetture dotate di calcolo vettoriale: somma HW di vettori di 4 elementi.

Architettura SIMD (Single Instruction – Multiple Data) o **Architettura vettoriale**:

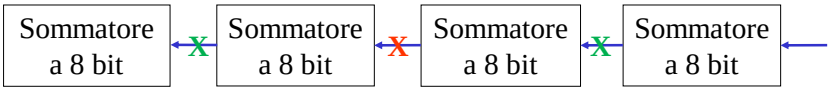
- Reduced bandwidth in fetch and decoding.
- Easy synchronization.
- Richiedono registri con ampiezza pari alla dimensione del vettore.

Modalità di calcolo vettoriale

- Operazioni sui vettori richiedono la stessa operazione su elementi adiacenti (multi-media, grafica, calcolo strutturale…)
- Multimedia have short data (gray levels: 8 bit, audio: 16 bit).
- Developed in Intel since 1996 as extension of the ISA: MMX (Multi-Media-Extension), SSE (Streaming-SIMD-Extension), AVX (Advanced-Vector-Extension), supported by HW.
- In ARM architecture, NEON extension to ISA has been developed to support vector processing.

- Vettore di dati + comando operazione
- Registro HW che contiene tutti gli elementi del vettore. L'eseguono l'operazione su ogni coppia di elementi

- Struttura modulare flessibile: 1 somma a 32 bit, 2 somme a 16 bit, 4 somme a 8 bit… => Tagli della catena dei riporti sotto il controllo della reservation station.
- I dati avviati in esecuzione sui cammini di esecuzione sono su 128 / 256 bit oggi.



Parallelismo a livello di parola (o **sub-word parallelism**)

Come sfruttare il parallelismo a livello di parola in SSE / SSE2

Estensioni MMX e SSE (xmm, registri a 128 bit):

Trasferimento dati	Aritmetica	Comparazione
MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm	ADD {SS/PS/SD/PD} xmm, mem/xmm	CMP {SS/PS/SD/PD}
	SUB {SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	MUL {SS/PS/SD/PD} xmm, mem/xmm	
MOV {H/L} {PS/PD} xmm, mem/xmm	DIV {SS/PS/SD/PD} xmm, mem/xmm	
	SQRT {SS/PS/SD/PD} xmm, mem/xmm	
	MAX {SS/PS/SD/PD} xmm, mem/xmm	
	MIN {SS/PS/SD/PD} xmm, mem/xmm	

Different data quantities can be inserted into an xmm register (128 bit):
SS – Scalar, Signle precision FP: 1 operand on 32 bit
PS – Packed Single precision FP: 4 operands on 32 bit
SD – Scalar, Double precision FP, 1 operand on 64 bit
PD – Packed Double precision FP, 2 operands on 64 bit
A – 128 bit aligned in memory

NB Anche floating point

Sub-words vector operation in AVX (dati di pipeline da 256 bit)

***Single operation specifies more data inside xmm registers*.**

```
#include <x86intrin.h>
addpd %xmm0, %xmm4          # Somma 2 coppie di variabili a 64 bit (in xmm e xmm4)
                            # «pd» stands for packed double precision)
```

In 2011 ***Advanced Vector Exentsion*** (AVX) has been provided by Intel, with registers of 256 bit (internal registers **ymm**) and in 2015 AVX512 with registers of 512 bit (zmm)

```
#include <x86intrin.h>
vaddpd %ymm0, %ymm4         # Somma 4 coppie di variabili a 64 bit (in ymm e ymm4)
vaddpd %zmm0, %zmm4         # Somma 8 coppie di variabili a 64 bit (in zmm e zmm4)
```

It supports also operations on three registers.

```
vaddpd %ymm0, %ynmm1, %ymm4     # Somma 4 coppie di variabili a 64 bit:
                                # (ymm1+ymm4->ymm0)
```

Efficient use when FP variables, adjacent in memory are loaded into registers.

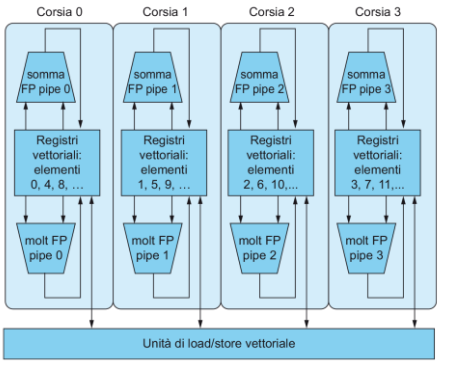
Vector architecture vs multi-media extensions

Width of elements is specified in a field (vector), it does not require a different opcode (SSE).

Data transfer is not required to be contiguous (vector: “gather-scatter” from memory).

Pipelined functional units (e.g. add + multiply fused together = pipelined inside each data processing path, both vector and SSE)

Multiple vectorial processing pathways, called **vector lanes** (cf. GPU processors).



Diversi tipi di parallelismo

Parallelismo (parziale) nell'esecuzione -> pipeline

Parallelismo nell'esecuzione su cammini multipli -> multiple issue

Parallelismo nell'elaborazione dei dati (vettoriali) -> parallelismo a livello di parola (sub-word parallelism)

Parallelismo su CPU diverse che condividono la memoria -> multicore

Sommario

Superpipeline
Multiple-Issue
Architetture SIMD