Pipelining

Performances

Qualora la pipeline non fosse uniforme, ovvero i tempi delle varie fasi sono diverse, non possiamo avere un **throughput minore del tempo che impiega lo stadio più lento** (quantità in unità di tempo).

Supponiamo che servano 100ps per leggere o scrivere un registro; gli altri stadi (memoria ecc) richiedono 200ps.

Cosa succede se mettiamo queste istruzioni in pipeline?

Istruzione di load

200ps fetch + 100 lettura reg + 200ps ALU op (per sommare la costante) + 200ps accesso alla memoria + 100ps scrittura registro finale.

Il totale è di 800 ps

Questa è l'istruzione più lenta.

Istruzione di store

200ps fetch + 100 lettura reg + 200ps ALU op (per sommare la costante) + 200ps accesso alla memoria.

Il totale è di 700ps

L'istruzione di store è uguale a quella di load, senza la scrittura finale in registro.

Istruzione R-Format

200ps fetch + 100 lettura reg + 200ps ALU op (per sommare la costante) + 100ps scrittura registro finale.

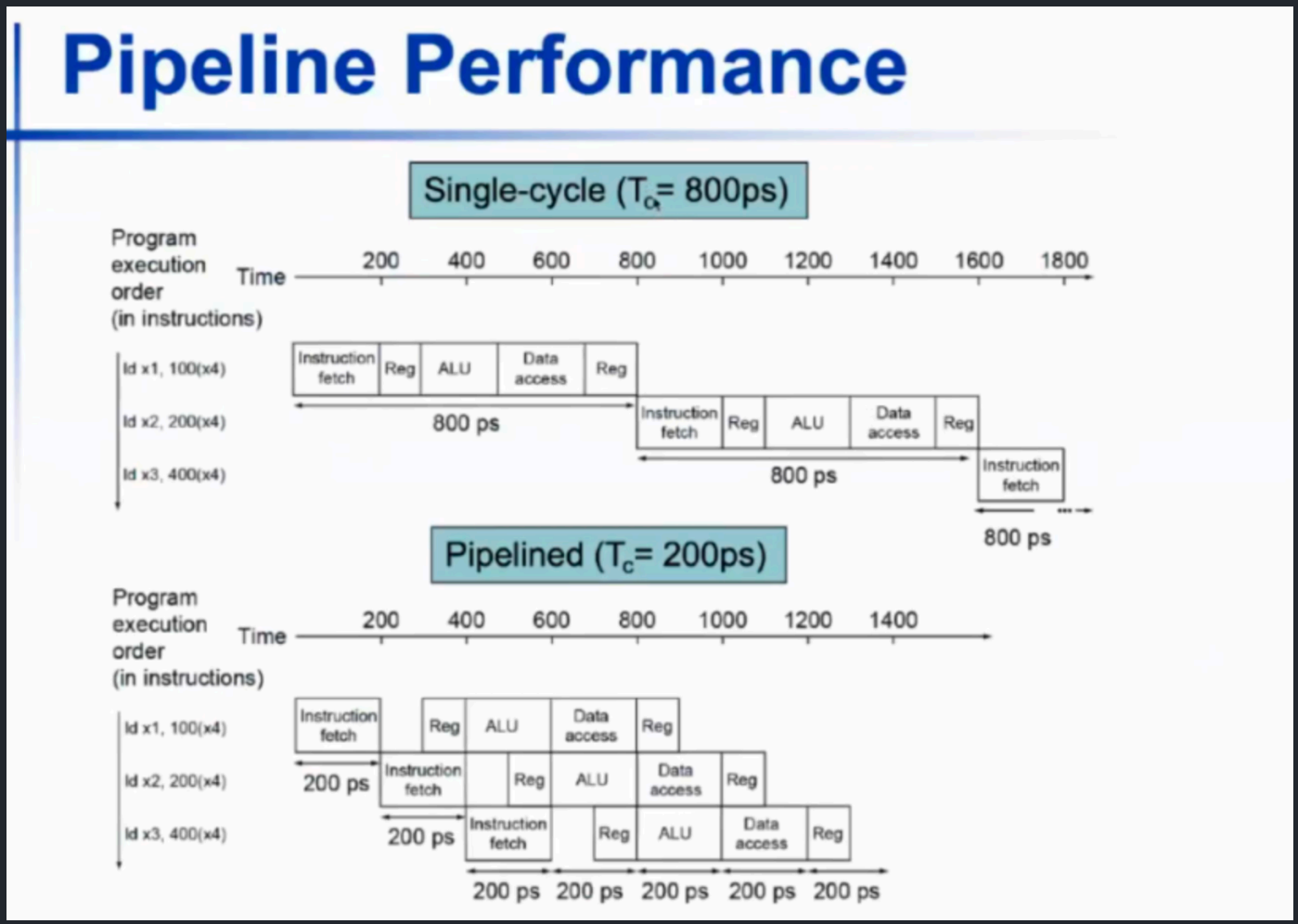
In questa istruzione non è presente l'accesso alla memoria, quindi il totale è di 600ps.

Istruzione di branch

200ps fetch + 100 lettura reg + 200ps ALU op (per sommare la costante).

Questa istr non effettua né l'accesso alla memoria nè la scrittura su registro; il totale è di 500ps.

Usiamo il pipelining con le precedenti istruzioni



In questa immagine notiamo che il T_c, ovvero il clock della macchina, è di 800ps. Questo perchè, come abbiamo detto, il clock deve adattarsi all'istruzione più lenta, ovvero quella di load, che dura proprio 800ps

Effettuiamo 3 load in sequenza

In questo caso, ovvero uno dei worst case possibili, se facessimo 3 ld in sequenza (senza pipeline) avremmo un tempo di esecuzione totale di 800 x 3.

Se utilizziamo il pipeline, invece, le istruzioni sono si squilibrate (quindi non durano tutte lo stesso tempo), ma possiamo lo stesso recuperare tempo.

In questo caso, il clock deve consentire alla **fase più lenta di un'istruzione** di essere eseguita, quindi nel nostro caso il clock deve essere di **almeno 200ps, quindi abbiamo un ulteriore vantaggio.**

Questo ci porta a dire che **ogni 200ps abbiamo una nuova fase eseguita**; non solo si va più veloce (per via del clock), ma anche un throughput maggiore!

Pipeline Speedup

Il caso migliore è quello in cui gli stadi sono **bilanciati**, ovvero durano tutte lo stesso tempo (non è questo il caso).

Tempo tra istruzioni$_{pipelined}$ = Tempo tra istruzioni$_{nonpipelined}$ / numero di stadi

Questa velocizzazione è dovuta ad un aumento del **throughput**.

La latenza, invece, ovvero il tempo per eseguire ogni istruzione, non diminuisce (ovviamente).

E' possibile convertire un vecchio processore per l'utilizzo del pipelining?

Se questa opzione non è stata presa in considerazione fin dall'inizio, non c'è molto da fare.

Se invece fin dall'inizio abbiamo un processore progettato con un set di istruzioni non troppo complicato, con tutte le istruzioni della stessa lunghezza, tutte destinate ad essere completate con un unico ciclo (ad esempio MIPS e RISCV), **l'implementazione è fattibile** e sopratutto semplice (relativamente).

Pipelining e RISC V

Il RISC è stato sicuramente progettato con in mente l'idea del pipelining, perchè avere un clock di 4 MHz nel 2000 serve relativamente a poco.

Ad esempio, come abbiamo visto, il RISC non può eseguire direttamente dalla RAM, ma utilizza la tecnica del load/store, quindi due istruzioni invece di una. Questo è un "cavallo di battaglia", proprio perchè avere due accessi alla memoria (in un'unica istruzione), **impedisce il pipelining**.

Hazards

Un hazard è una situazione rischiosa in generale; sono delle situazioni in cui dei circuiti progettati male potrebbero fornire dei risultati errati.

In questo contesto gli **hazards** sono delle situazioni particolari che impediscono che l'istruzione parta **precisamente** al prossimo ciclo di clock.

Questi hazards, pongono dei limiti al pipelining, ed a volte, impediscono che si possa procedere all'esecuzione della prossima esecuzione, come dovrebbe accadere in uno schema **teorico** come quello visto precedentemente.

Gli hazards che si possono verificare nelle pipeline sono suddivisi in 3 categorie:

- Hazard sulle strutture: sto tentando di eseguire qualcosa, ma non posso farlo perchè una risorsa che mi serve è occupata.
- Hazard sui dati
- Hazard di controllo

🏁 0:37 04-9

Hazard di strutture

Il RISC ha due memorie: **Memoria istruzioni e memoria dati**.

Se la memoria fosse unica per istruzioni e dati, mentre effettuo il fetch di un'istruzione, sicuramente non posso scrivere in memoria!

Cosa succede se, arrivato ad un certo punto della pipe, dovrei scrivere in memoria ma questa è occupata? Bisogna quindi **bloccare la pipe**, e non posso procedere alla fase successiva, perchè devo attendere che la memoria (o altro) si liberi.

Questo avvenimento si chiama **stall**.

Quando abbiamo una sola memoria, e non possiamo quindi avere un sistema di pipelining, il processore deve essere dotato di almeno **due cache**.

Hazard di dati

```
add x19, x0, x1
sub x2, x19, x3
```

In questo codice notiamo che il registro x19 è un operando sia nella prima che nella seconda istruzione;

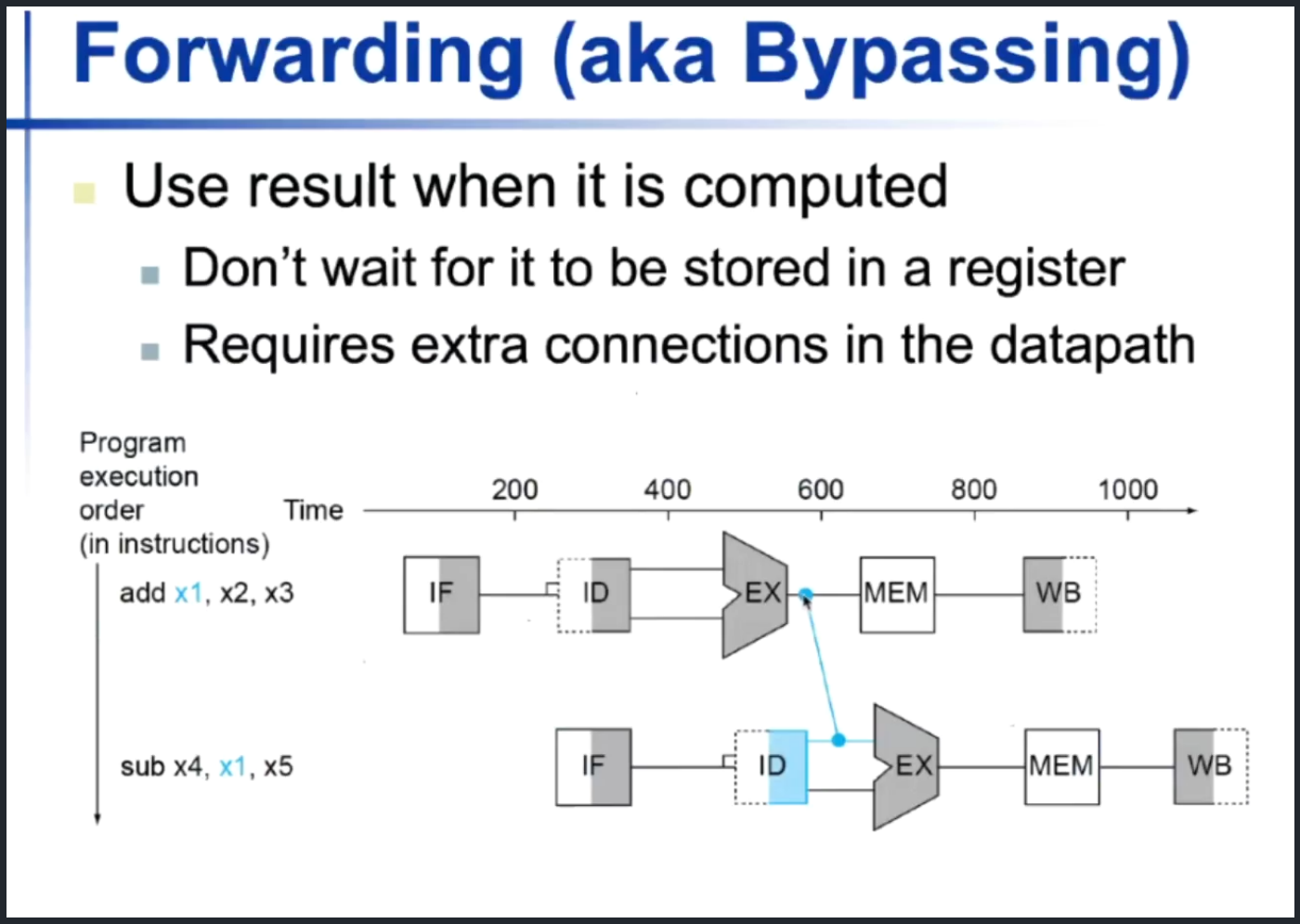
Questo significa che finchè non scrivo in x19 il risultato, non posso sottrarci x3. Si capisce immediatamente che il fatto che ci sia, nella seconda istr, un operando sorgente che dipende dall'istr precedente, fa sì che il valore di x19 non sarà disponibile finchè l'istruzione precedente non sarà completata.

Di conseguenza, non possiamo far avvenire queste istruzioni in pipeline.

In gergo, **ritardare la pipe** si dice "**introdurre una bolla**".

Soluzione : Forwarding

Esiste una soluzione che risolve questo problema:



La soluzione sta proprio nel prendere **direttamente il risultato dell'ALU della prima istruzione** e portarlo come input all'ALU della seconda istruzione, senza dover attendere che la prima istruzione sia completa; in questo modo non dobbiamo nemmeno inserire il risultato nel registro (risparmiando tempo).

Questo però ha un costo hardware, proprio perchè dobbiamo inserire un altro collegamento all'interno del **datapath** del processore;

Questa nuova connessione parte dall'output dell'ALU e torna all'Input dell'ALU stessa.

🏁 1:00

Scheduling del codice per evitare stalli

Vogliamo calcolare: `a = b + e; c = b + f`.

Se traduciamo questa operazione in assembly senza pensarci troppo, useremo tre operazioni di load e due di store:

```
ld x1, 0(x0)
ld x2, 8(x0)

add x3, x1, x2
sd x3, 24(x0)
ld x4, 10(x0)
add x5, x1, x4
add x5, x1, x4

sd x5, 32(x0)
```

> 13 cicli di clock

Inevitabilmente abbiamo due stalli, uno per ogni operazione di add; questo perchè dobbiamo caricare una volta nel registro x2 ed una volta nel registro x4, e poi usare i registri come operandi nell'istruzione di add.

Codice che evita gli stalli

```
ld x1, 0(x0)
ld x2, 8(x0)
ld x4, 16(x0)

add x3, x1, x2
sd x3, 24(x0)

add x5, x1, x4

sd x5, 32(x0)
```

> 11 cicli di clock

Con questo codice, abbiamo leggermente riorganizzato le operazioni di load: se effettuiamo prima le operazioni di load, in modo che mentre il calcolatore carica x4, posso iniziare ad effettuare l'add con x2.

Morale della favola

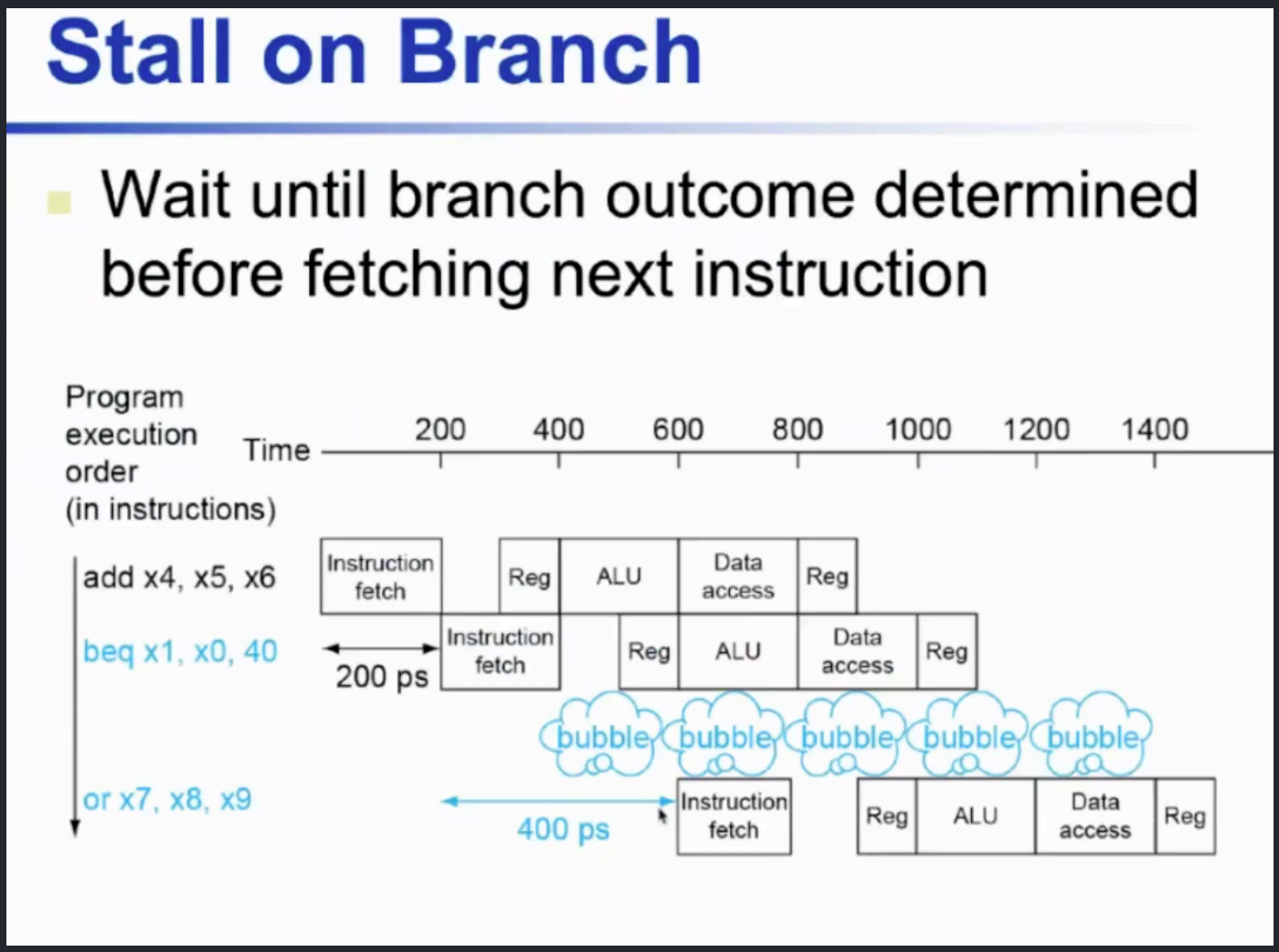
Dove l'hardware non riesce ad ottimizzare le sue operazioni, sta al programmatore scrivere del codice ottimizzato affinché il programma sia efficiente.

Hazard di controllo

Gli hazard di controllo sono legati ai **branch**. Il RISC dispone sia di istruzioni di jump, quindi indipendentemente dal resto saltano ad un'altra parte di codice.

Il problema sono i **salti condizionati** (branch if ...), proprio perchè il salto dipende dal risultato del branch. Questo significa che finchè **l'outcome** non è stato calcolato (dall'ALU), non è possibile "saltare".

Stallo sul branch



Se aggiungiamo al processore un minimo di hardware che possa confrontare i registri, è possibile il numero di **bolle**

Come risolvere?

L'idea è quella di **prevedere** il risultato del branch.

Ovviamente non si può tirare ad indovinare, ma questa tecnica può essere usata per prevedere il risultato di un **branch usato per creare un loop**.

Con quest'idea, possiamo assolutamente formulare un'ipotesi, e quindi prevedere se il branch effettuerà un jump o meno.

Quindi, il processore prevede se andare avanti o meno; se prevede di andare avanti, va avanti **prima ancora che l'alu abbia risposto**; se l'ALU, una volta calcolato, dice di andare avanti, il processore ha previsto bene. Se invece l'ALU dice che si doveva effettuare il salto, il processore deve **cancellare il risultato delle istruzioni eseguite fino a quel punto** e riprendere l'esecuzione dal jump.

Previsione del salto dinamicamente

Questa tecnica non viene usata dai processori moderni, perchè azzeccare il salto solo nel 60% dei casi non è accettabile.

Viene invece usata una previsione **dinamica**; se sono all'interno di un loop ed il salto è stato previsto, **è probabile** che la prossima volta il salto verrà previsto nuovamente; quindi ricordando cosa è successo precedentemente per quel particolare branch;

Con questo sistema il livello di casi corretti aumenta sostanzialmente.

L'hardware, ovviamente, deve essere predisposto in modo tale che nel caso in cui un branch viene previsto in maniera errata, si possa immediatamente tornare indietro; tutti i risultati intermedi devono quindi essere tenuti in cache e **non essere scritti in memoria**, in modo da fare un **rollback** in caso necessario.

Riassunto pipeline

La tecnica del pipelining serve a sovrapporre nel tempo i vari stadi di un'operazione, in modo da tenere sempre occupata ogni sezione che serve a risolvere un dato stadio.

Questa tecnica viene però frenata dalla presenza del problema **degli hazards**, che possono essere di **struttura, data, o controllo**;

Fine lezione 13