Programmazione Avanzata e parallela

Lezione 05

Luca Manzoni

Cosa vedremo oggi

Pipelines, branch prediction e branchless programming

- Cosa è una pipeline
- Esecuzione superscalare e out-of-order
- Pipeline hazards
- Branch prediction e il suo costo
- Branchless programming

Alcune definizioni

Latenza e throughput

- **Latenza** (*latency*): il tempo che intercorre da quando una operazione inizia e quando viene completata
- La latenza può essere misurata in termini di tempo (quanti nanosecondi) ma più normalmente in numero di cicli perché una istruzione completi
- **Throughput**: il numero di operazioni completate per unità di tempo
- Il throughput può essere misurato in termini di **IPC** (*instructions per cycle*): quante istruzioni sono completate a ogni ciclo di clock

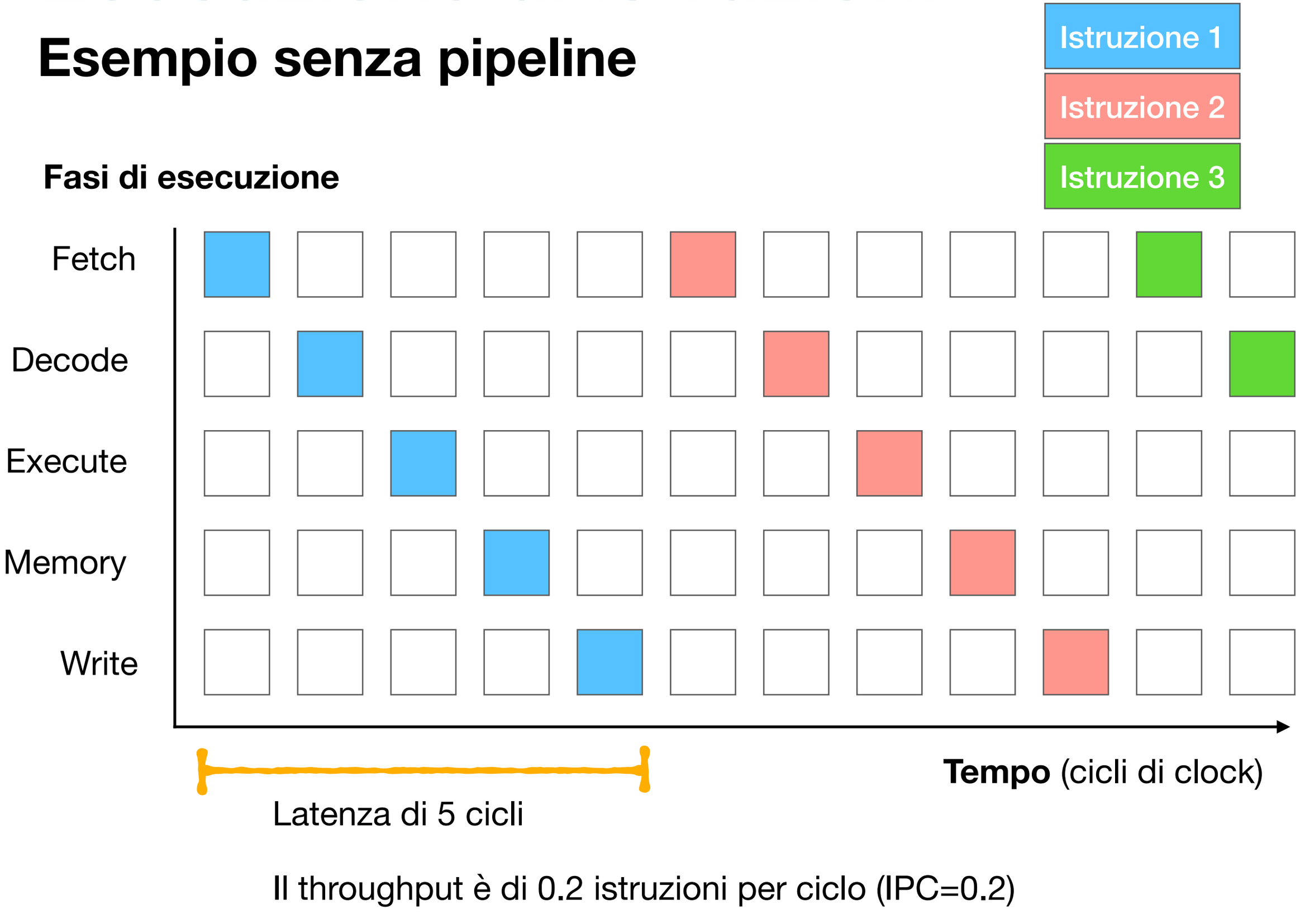
Fetch-decode-execute revised

Qualche dettaglio aggiuntivo

- Schema “base” di come viene eseguita una istruzione:
  - **Fetch**. L’istruzione viene caricata dalla memoria
  - **Decode**. L’istruzione viene decodificata
  - **Execute**. L’istruzione viene eseguita, facendo magari uso della ALU o della FPU (floating-point unit) e magari con…
  - **Memory**. Accesso alla memoria
  - **Write**. Infine i risultati devono essere scritti nei registri

Esecuzione di istruzioni

Esempio senza pipeline



Il throughput è di 0.2 istruzioni per ciclo (IPC=0.2)

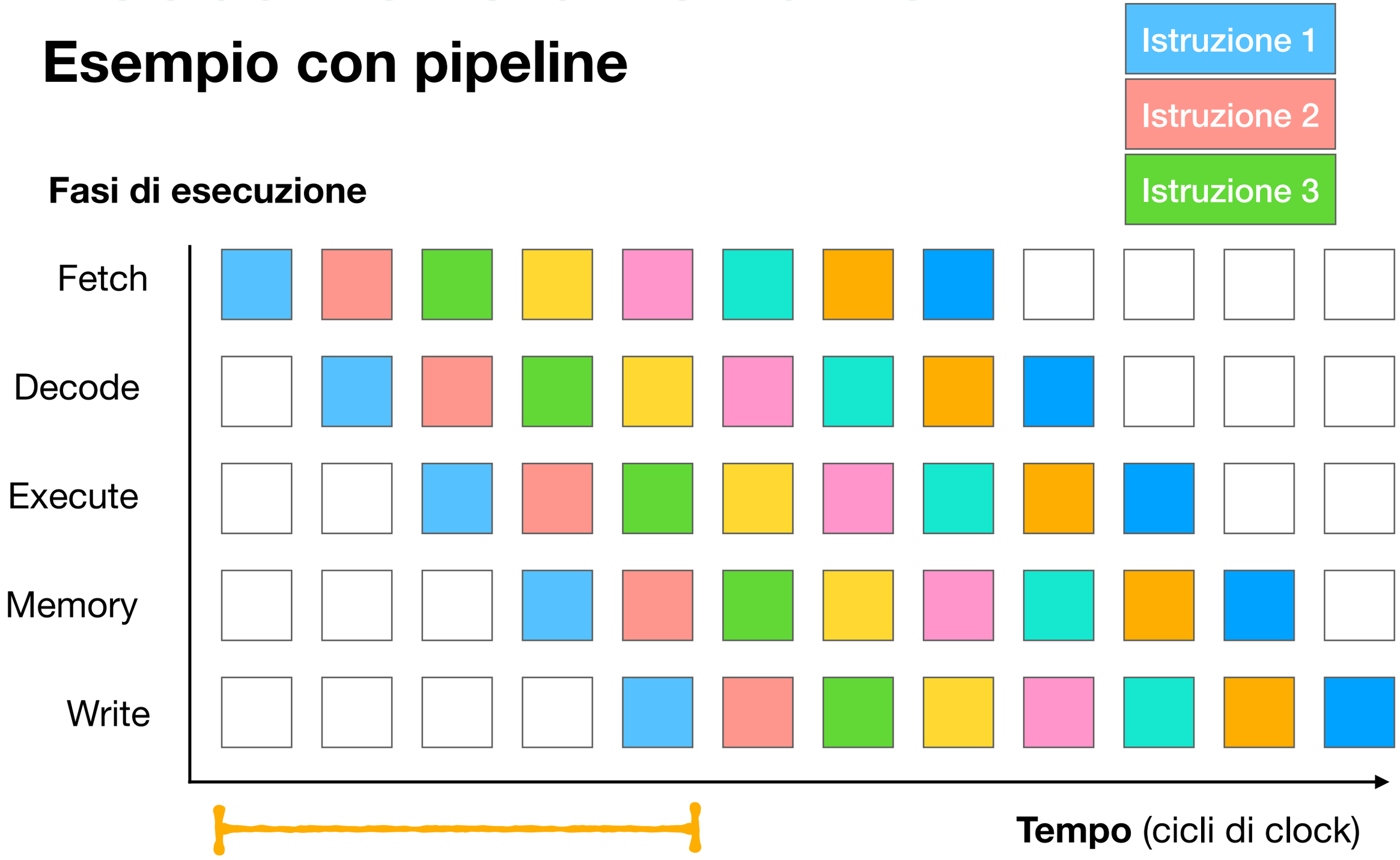
Alcune osservazioni

Perché pipelines?

- Notiamo che queste fasi possono essere compiute da diverse componenti della CPU
- La decodifica non userà la ALU, la ALU non userà l'unità di load/store, etc.
- Se sappiamo quale è la prossima istruzione da eseguire allora possiamo iniziare l'esecuzione prima che quella precedente abbia completato
- Questo non aiuta la latenza ma aiuta il throughput (domanda: perché?)

Esecuzione di istruzioni

Esempio con pipeline



Il throughput è maggiore. Di quanto?

Alcune osservazioni

Perché pipelines?

- All’inizio le istruzioni iniziano a eseguire una alla volta fino a che tutte le diverse componenti non sono impegnate a eseguire qualcosa. Questo è il setup cost.

- Successivamente viene completata una istruzione per ciclo di clock e una nuova istruzione inizia l’esecuzione

- Quindi, se abbiamo molte istruzioni possiamo ottenere un valore di IPC molto vicino a 1

- Nei processori moderni ci sono molti più stadi di pipeline e le istruzioni possono richiedere un numero diverso di cicli per eseguire

Esecuzione superscalare

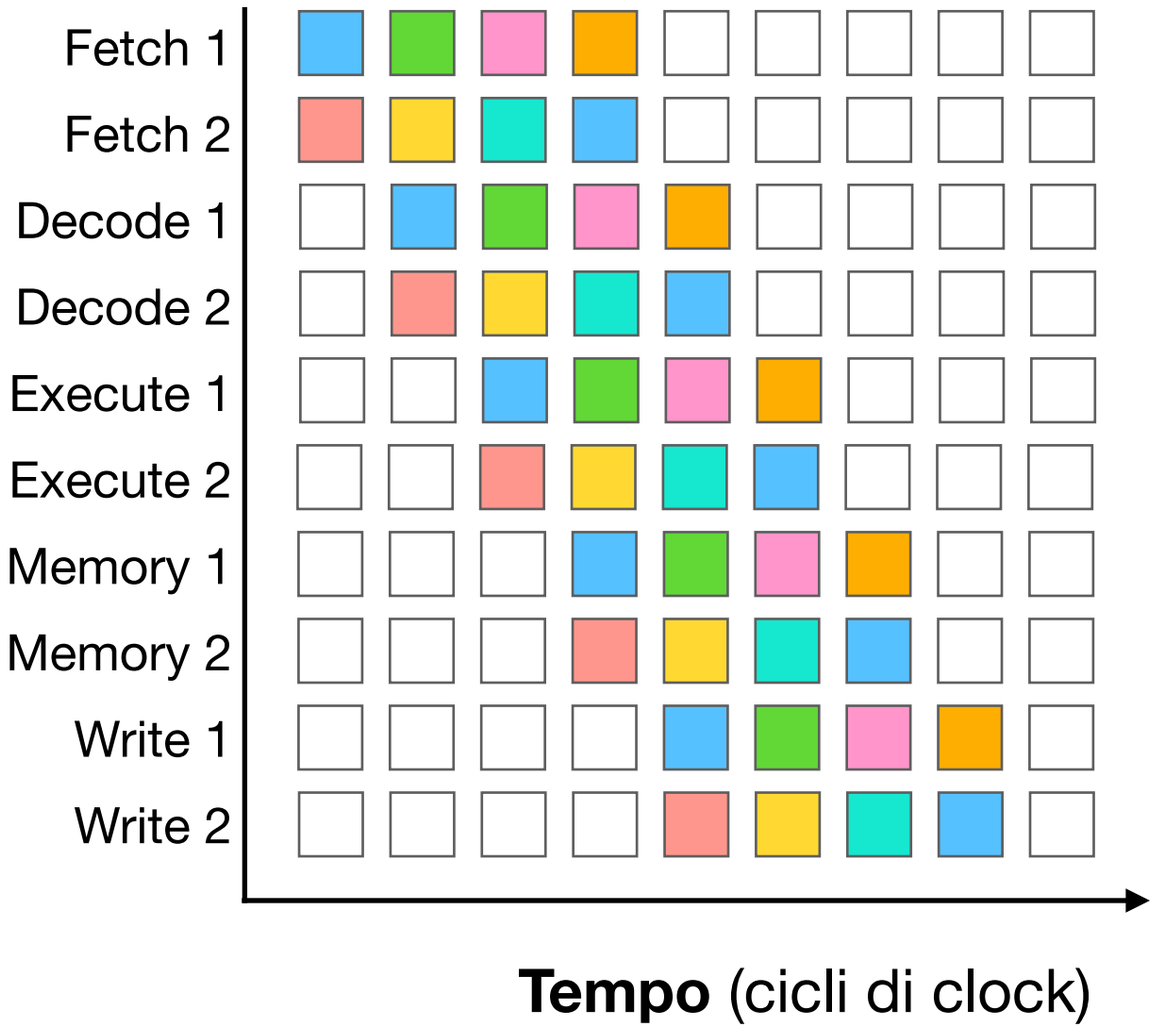
E instruction-level parallelism

- Se abbiamo due istruzioni consecutive che operano su dati differenti (e.g., registri differenti) potenzialmente possono essere eseguite in parallelo…
- …se ci sono le risorse per eseguirle (e.g. ALU, decodifica, etc)
- Questo ci permette di sfruttare del parallelismo all'interno delle istruzioni che che dobbiamo eseguire
- Richiede avere molte più componenti duplicate assieme a componenti in grado di decidere se due istruzioni possono essere eseguite in parallelo

Esecuzione di istruzioni

Esecuzione superscalare

Fasi di esecuzione



Se possiamo sempre eseguire due istruzioni in parallelo allora il throughput aumenta

In particolare possiamo ottenere un IPC > 1 completando più di una istruzione per ciclo di clock

Ma se due istruzioni consecutive non sono eseguibili in parallelo?

Latenza e throughput

Il caso delle CPU moderne

- Per le CPU moderne il tempo di esecuzione di ogni istruzione può variare (e a volte, per la stessa istruzione, varia a seconda degli operandi)
- Cambia anche quante istruzioni di quel tipo posso essere eseguite nello stesso momento
- Una misura per questo è il Reverse Throughput (RThroughput), che con un valore < 1 significa che più istruzioni possono essere eseguite nello stesso momento
- Tutte queste informazioni sono collezionate in documenti specifici detti “instruction tables”

Instruction tables

Esempio per Zen 4

Tipo degli operandi

Latenza

RThroughput

Operazioni

Arithmetic instructions				
ADD, SUB	r,r	1	1	0.25
ADD, SUB	r,i	1	1	0.25
ADD, SUB	r,m	1		0.33
ADD, SUB	m,r8/16	2	7-8	1
ADD, SUB	m,r32/64	2	1	1
ADC, SBB	r,r	1	1	1
ADC, SBB	r,i	1	1	1
ADC, SBB	r,m	1	1	1
ADC, SBB	m,r8/16	2	8	1
ADC, SBB	m,r32/64	2	1	1
ADCX ADOX	r,r	1	1	1
CMP	r,r	1	1	0.25
CMP	r,i	1	1	0.25
CMP	r,m	1		0.33
CMP	m,i	1		0.33
INC, DEC, NEG	r	1	1	0.25

https://www.agner.org/optimize/instruction_tables.pdf

Esecuzione out-of-order

E instruction-level parallelism

- Anche se possiamo eseguire fino a n istruzioni in parallelo duplicando l'hardware non è detto che n istruzioni consecutive siano sempre eseguibili in parallelo
- Possiamo però cambiare l'ordine delle istruzioni:
  - Le istruzioni possono iniziare a eseguire nel momento in cui hanno gli operandi sono disponibili
  - Dobbiamo però scrivere i risultati nell'ordine giusto
- I moderni processori implementano questo riordinamento delle istruzioni (esecuzione out-of-order)

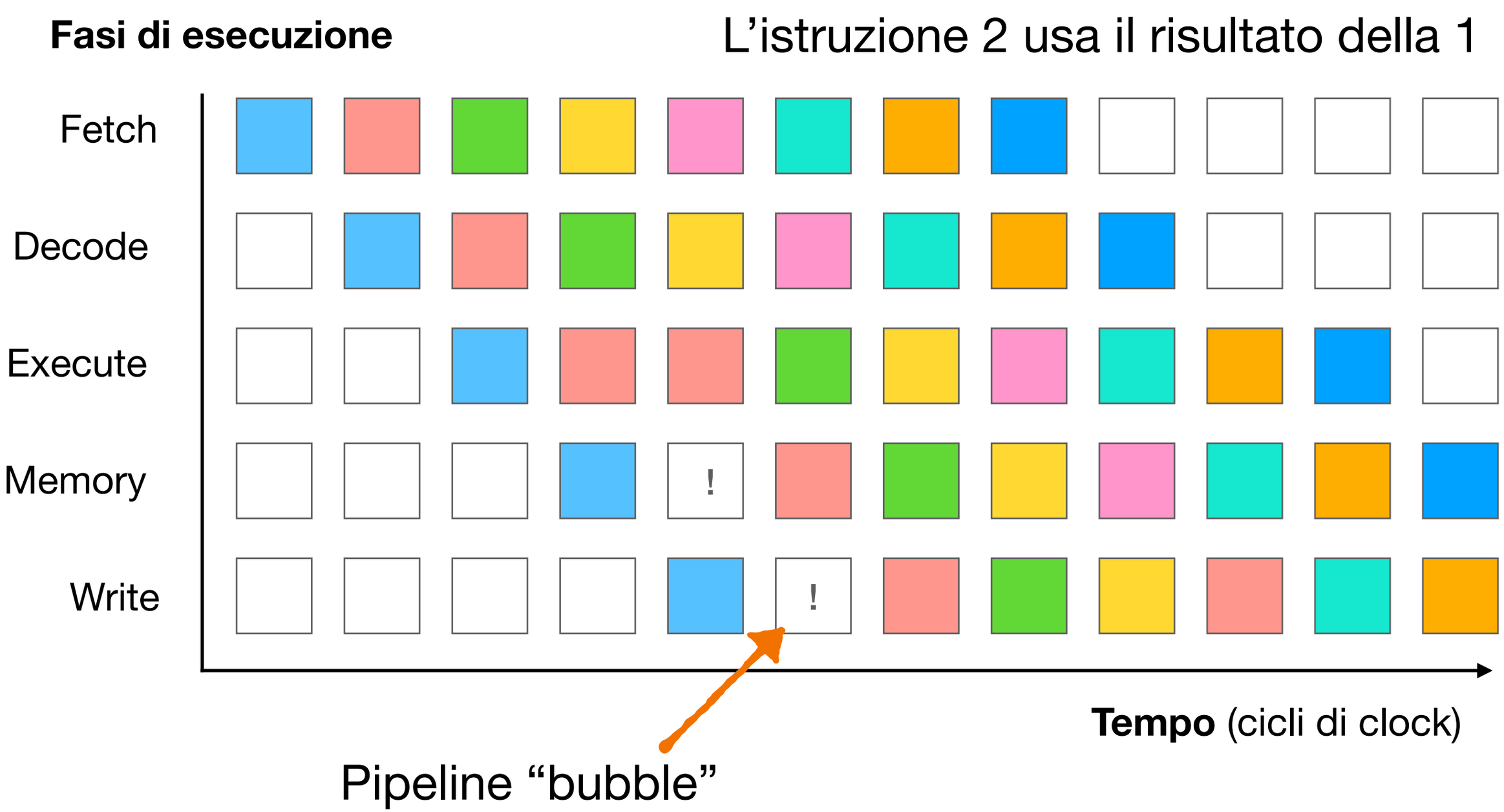
Esecuzione out-of-order

Il ciclo fetch-decode-execute

- **Fetch**. L'istruzione viene caricata dalla memoria
- **Decode**. L'istruzione viene decodificata
- **Dispatch**. L'istruzione viene inserita in una coda di istruzioni da eseguire
- Quando tutti gli operandi per eseguire l'istruzione sono presenti l'istruzione può iniziare l'esecuzione (anche prima di istruzioni precedenti)
- I risultati sono inseriti in una coda
- **Retire**. Quando tutte le istruzioni precedenti hanno scritto i risultati possiamo scrivere i risultati

Esecuzione di istruzioni

Pipeline bubbles



Pipeline hazards

Quando dobbiamo bloccare la pipeline

- Possiamo identificare tre casi in cui la pipeline deve aspettare:
  - **Structural hazard**. Quando due o più istruzioni necessitano della stessa parte della CPU (e.g., la ALU)
  - **Data hazard**. Quando una istruzione necessita di un operando che deve essere ancora computato
  - **Control hazard**. Quando non è possibile stabilire la successiva istruzione da eseguire
- In tutti questi casi vengono inserire delle “bolle” (*bubbles*) nella pipeline in cui non viene eseguito nulla

Pipeline hazards

Penalità

- Le penalità per i diversi hazard sono differenti:
  - **Structural hazard**. Dobbiamo attendere che l'unità si liberi (solitamente 1-2 cicli), queste penalità dipendono dall'hardware a disposizione
  - **Data hazard**. Dobbiamo attendere che il risultato sia disponibile (la latenza del "*critical path*"), possiamo ristrutturare la computazione per limitare questa penalità
  - **Control hazard**. In questo caso non possiamo eseguire nessuna altra istruzione fino a quando non sappiamo la prossima istruzione da eseguire, solitamente perdendo 15-20 cicli (e poi dovendo tornare a riempire la pipeline)

Esecuzione speculativa

E branch prediction

- Dato che i control hazard derivati dai branch sono costosi è possibile pensare a una esecuzione speculativa
- In un branch possiamo avere due (o più) target in cui la computazione prosegue
- Possiamo sceglierne un target (i.e., la prossima istruzione da eseguire) e proseguire da lì tenendo “in sospeso” i risultati
- Questa è l’*esecuzione speculativa*, in cui assumiamo di conoscere la prossima istruzione e proseguiamo nell’esecuzione
- Questo ci permette di non pagare sempre la penalità per un control hazard

Esecuzione speculativa

E branch prediction

- Una volta che sappiamo dove l'esecuzione deve proseguire abbiamo due possibilità:
  - *La scelta era corretta*. In questo caso possiamo effettivamente scrivere i risultati senza pagare la penalità
  - *La scelta era sbagliata*. Dobbiamo scartare i risultati e proseguire l'esecuzione dall'istruzione corretta. In questo caso non abbiamo risparmiato nulla
- Il modo in cui facciamo la scelta di dove proseguire è dettato dal sistema di **branch prediction** che abbiamo

Esecuzione speculativa

E branch prediction

- Un branch predictor generalmente tiene delle statistiche sui branch presi/non presi e le usa per la predizione. Alcuni esempi:
- *Branch predictor statico*. Non tiene statistiche, sceglie sempre preso o non preso
- *Branch predictor con contatore a saturazione.* Usando un contatore di n bit aggiungendo $+1$ se il branch è preso e -1 se non preso (fino a un minimo e un massimo), si decide la predizione in base ad una soglia.

  Nel caso di 1 bit corrisponde a fare quello che è successo l'ultima volta che si è incontrato il branch

Esecuzione speculativa

E branch prediction

- *Branch predictor a due livelli*. Si utilizza lo storico di n bit (e.g., 010 per non-preso, preso, non-preso) per indicizzare in una tabella il predittore apposito per quello storico (e.g., uno con contatore a saturazione)

- *Branch predictor locale*. Predittore separato per i diversi branch (e.g., in base all'indirizzo del branch), magari uno a due livelli.

- *Branch predictor globale*. Predittore globale che usalo storico di tutti i branch (utile per vedere correlazioni tra i branch)

- Possibilità di combinare più branch predictors in un sistema ibrido che sceglie tra più predittori

- Predatori specifici per cicli, ritorni da funzione e branch indiretti

- … tanti altri

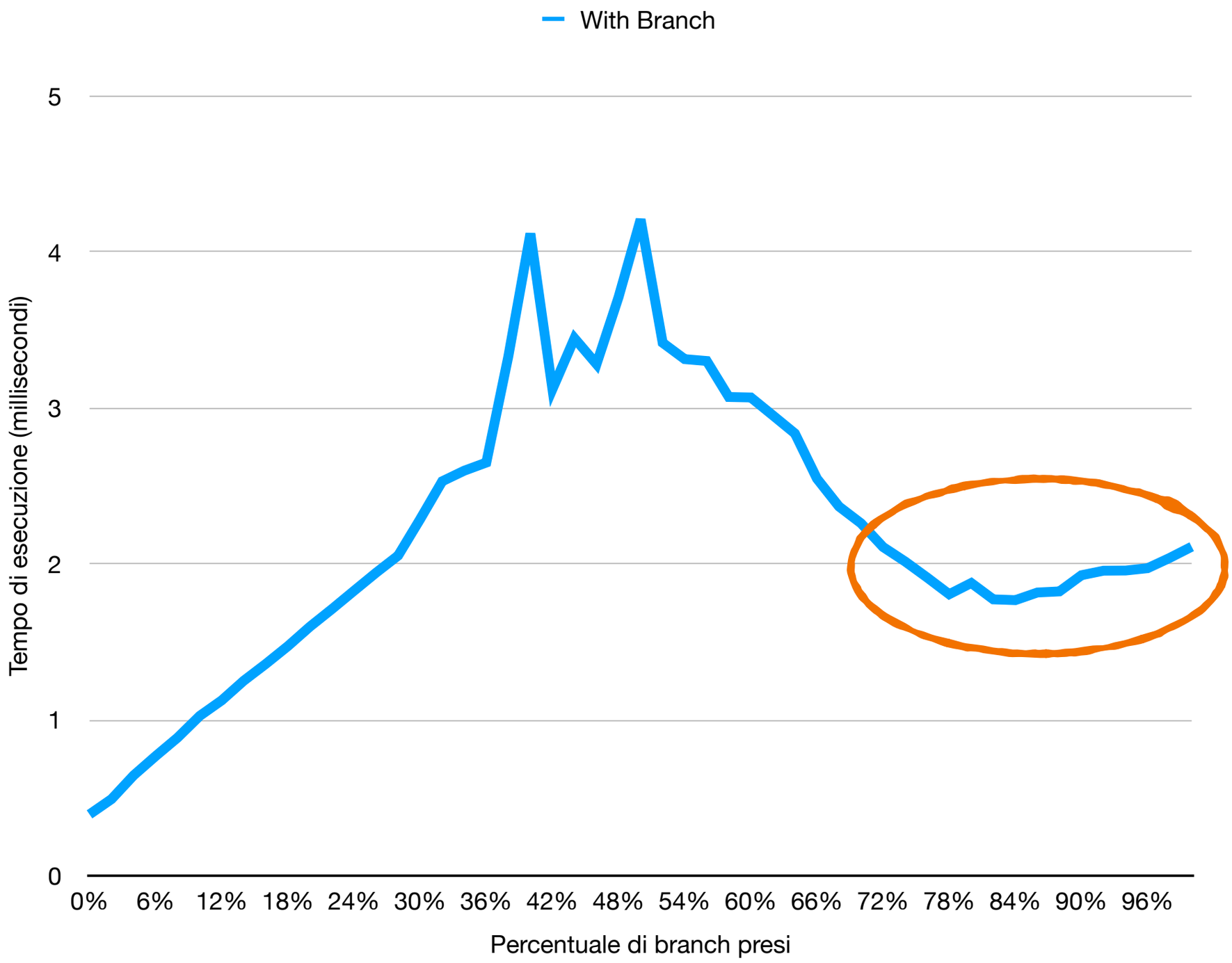
Costo della predizione sbagliata

Ed effetto del branch predictor

- Possiamo misurare il costo di una predizione sbagliata in modo semplice:
  - Generiamo un array di numeri casuali tra 0 e 99
  - Scegliamo una soglia $0 \leq p \leq 1$
  - Sommiamo l'i-esimo elemento dell'array solo se $v[i] \leq 100p$
  - In questo modo (essendo numeri casuali) anche il predittore migliore non potrà far giusto con probabilità maggiore di $\max\{p, 1-p\}$ (ovvero predizione "sempre preso" o "sempre non preso")

Il costo dei branch

Effetto sul tempo di esecuzione



Il grafico non è simmetrico perché comunque quando il branch è (quasi) sempre preso abbiamo più operazioni (le somme) da eseguire

Costo della predizione sbagliata

Ed effetto del branch predictor

- Se i dati fossero ordinati ci sarebbe un pattern tra due branch successivi:
  - “Se hai preso il precedente prendi anche il successivo” è vero sempre tranne una volta
  - “Se non hai preso il precedente non prendere il successivo” è vero sempre
- I branch predictor sono molto buoni per questo tipo di pattern, quindi la predizione sarà quasi sempre corretta

Programmazione branchless

E quando è utile

- In alcuni casi è possibile evitare i branch usando delle tecniche per programmazione branchless

- Se dobbiamo scegliere tra due valori a e b a seconda di una condizione possiamo sfruttare due fatti:

  - Dato $q \in \{0,1\}$, abbiamo che $aq + (1 - q)b$ può assumere solo due valori:

    - a quando $q = 1$

    - b quando $q = 0$

  - Condizioni come $x \leq 10$ in C ritornano un valore che è 0 o 1

Programmazione branchless

E quando è utile

- Quindi con $q = v[i] < 100p$ otteniamo un valore che è 0 o 1 a seconda della condizione

- Con $q * v[i] + (1 - q) * 0$ otteniamo il valore corretto da sommare

- In questo modo non abbiamo più un branch…

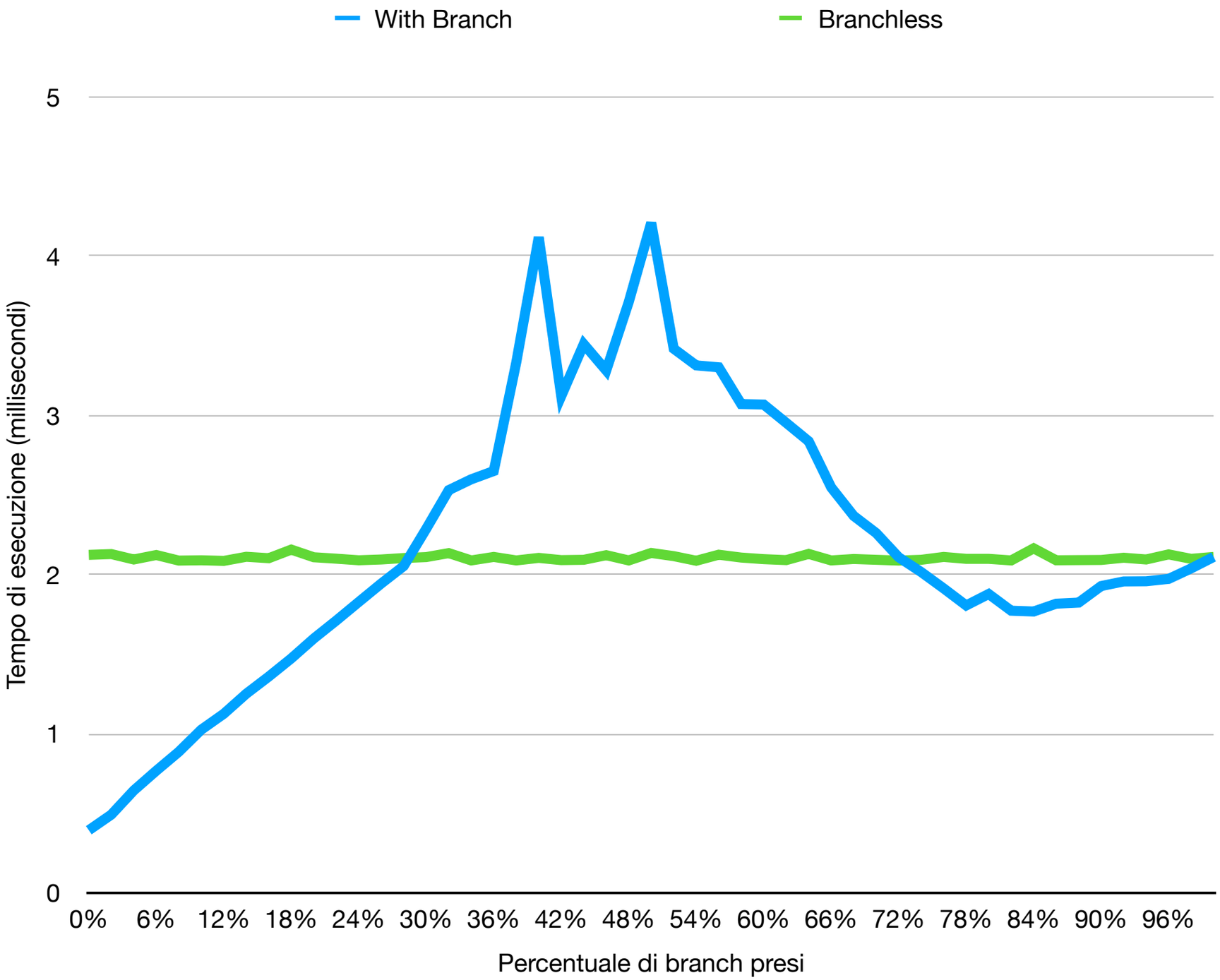
- …ma lo abbiamo rimpiazzato con un *data hazard*.
  La moltiplicazione ora dipende dalla comparazione precedente

- Inoltre, adesso eseguiamo sempre delle operazioni aritmetiche anche quando non ci servono

- Quanto ci guadagniamo?

Programmazione branchless

Effetti e guadagni



Programmazione branchless

E quando è utile

- Questa tecnica è generalmente chiamata “predication”
- Possiamo computare entrambi i lati di un “if” e poi scegliere quale tenere usando solo operazioni aritmetiche
- È valida se computare entrambi i lati dell’“if” è meno costoso che computare solo quello giusto sbagliando però nell’esecuzione speculativa una certa frazione delle volte…
- …ovvero quando si fanno poche operazioni per ogni lato dell’“if” e il branch è di difficile predizione
- Conviene eseguire dei test per verificare di volta in volta

Programmazione branchless

Scelta automatica

- Un compilatore può (potenzialmente) scegliere di utilizzare operazioni che fanno “predication”: in x86-64 c’è *cmov*, armv8-a c’è *csel*

- Vengono utilizzate una serie di euristiche su quando sia meglio…

- …ma possiamo usare la *profile-guided optimization (PGO)* per prendere statistiche che aiutano il compilatore a scegliere che istruzioni generare

- In generale istruzioni che lavorano su vettori di dati (e.g., SIMD che vedremo più avanti) sfruttano branchless programming…

- …e infatti se negli esempi rimuoviamo “volatile” il codice avrà lo stesso tempo di esecuzione in ogni caso