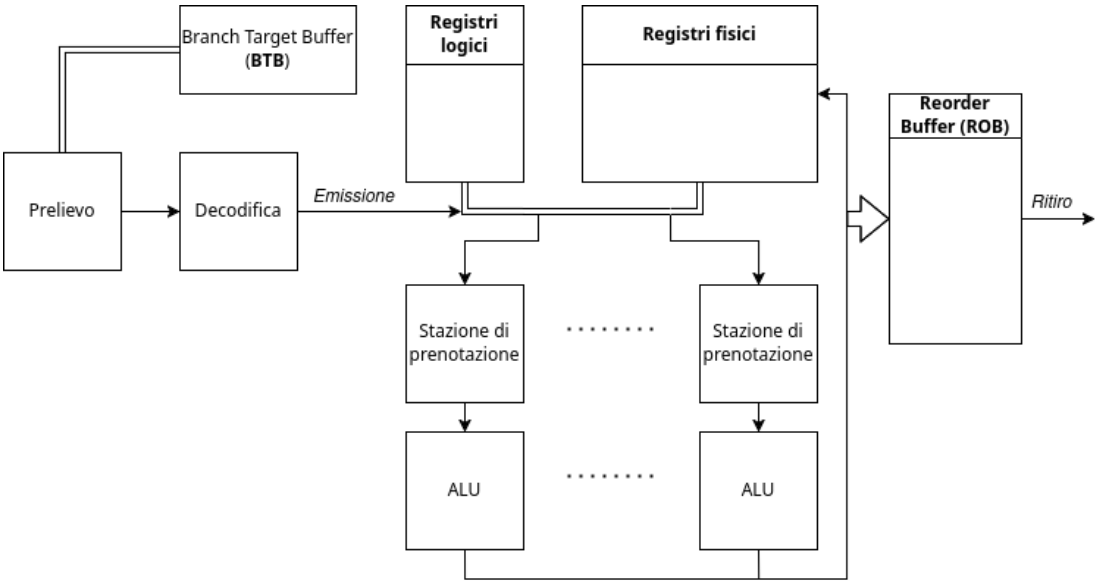
1 Lezione del 09-05-25

Riprendiamo il discorso sull'architettura del processore. Possiamo fare un riassunto dell'architettura descritta finora:



Abbiamo quindi che le istruzioni provengono dal Branch Target Buffer, **BTB**, dalle quali vengono poste in una componente di **prelievo** e una di **decodifica**.

Da qui i poi le istruzioni passano per le stazioni di prenotazione, che poi le inviano alle relative ALU, sulla base delle tabelle associate ai registri fisici.

In fine, in fondo a questo meccanismo si trova il Reorder Buffer, **ROB**, che contiene le istruzioni terminate ma non ancora scritte e permette l'esecuzione speculativa. Il significato del Reorder Buffer è sostanzialmente quello di realizzare un'ulteriore smistamento delle istruzioni a termine della loro esecuzione, che permette o meno il loro write back nei registri (che sappiamo coi registri ridenominati significa semplicemente aggiornare i registri logici) solo sotto le condizioni dell'esecuzione speculativa.

1.1 Ritiro di istruzioni operative

Quanto appena detto, riguardo alle istruzioni operative, viene eseguito memorizzando 2 registri fisici per registro logico:

- Il registro fisico **speculativo**, contenente il registro fisico associato dall'ultima istruzione emessa ma non ritirata;
- Il registro fisico **non speculativo**, contenente il registro fisico associato dall'ultima istruzione ritirata.

Si ha quindi che in caso di ritiro corretto di un istruzione, si aggiorna il registro destinazione con il registro speculativo, mentre se l'istruzione non viene ritirata si mantiene il registro non speculativo.

1.2 Ritiro di istruzioni di accesso in memoria

Per quanto riguarda le istruzioni di accesso in memoria, vorremo evitare innanzitutto le dipendenze, come quelle viste riguardo ai registri.

Per fare ciò possiamo fornirci di un buffer dedicato alle istruzioni di scrittura in memoria (**store**), detto **Store Buffer**. L'unicità di tale buffer basterà da sola a evitare le dipendenze in uscita.

Per le istruzioni di lettura in memoria (**load**) potremo invece usare più **Load Buffer**, in quanto non ci sono dipendenze sulle letture parallele.

Nei buffer Store e Load manteniamo due campi: l'indirizzo del dato e il valore a tale indirizzo. Questo può essere precalcolato per le letture (si possono anticipare le LOAD, ed è questo che velocizza di molto le prestazioni, ad esempio anticipando cacheline non caricate), mentre per le scritture bisogna calcolare il risultato da scrivere e aspettare fino al ritiro per effettuare la scrittura effettiva.

Abbiamo quindi che:

- Le dipendenze in uscita possono essere risolte controllando, al momento della lettura, che tutte le scritture precedenti sugli stessi indirizzi siano state completate (nel caso l'ultimo valore sia già stato calcolato, si può inoltre inoltrare il risultato direttamente alla load, attraverso il cosiddetto *Store Forwarding*);
- Le antidipendenze possono essere risolte con la metodologia opposta, cioè controllando, al momento della scrittura, che tutte le letture precedenti sugli stessi indirizzi siano state completate.

A questo punto le alee date dall'esecuzione speculativa si risolvono semplicemente ritardando le scritture fino alla fase di ritiro dal ROB (come già facevamo), e svuotando il ROB in caso di predizioni sbagliate (come già facevamo), effettivamente annullando le letture. Questo approccio, per quanto funzionale dal punto di vista delle prestazioni, presenterà però delle falle di sicurezza che vedremo nei prossimi paragrafi.

1.2.1 Ritiro delle istruzioni di accesso allo spazio di I/O

Per quanto riguarda le istruzioni che comunicano con lo spazio di I/O, e quindi con le interfacce, non possiamo sfruttare l'esecuzione speculativa. Questo per via del fatto che le interfacce mantengono il loro stato, e non si comportano come la memoria tradizionale: le letture stesse potrebbero essere distruttive.

L'unica soluzione che abbiamo è quindi quella di fornire una sola unità di esecuzione alle operazioni sullo spazio di I/O, e assicurarci che queste vengano eseguite in ordine strettamente sequenziale.

1.3 Vulnerabilità Meltdown e Spectre

Meltdown e **Spectre** sono state vulnerabilità dell'architettura Intel x86, scoperte nel 2017, che permettevano di superare le limitazioni normalmente imposte al codice in esecuzione in modalità utente.

Abbiamo detto che il programma dovrebbe poter ignorare ciò che accade nel processore a livello **microarchitetturale**, cioè che tutte le operazioni che l'architettura descritta finora compie ai fini di ottimizzare l'esecuzione devono risultare **invisibili** al programma.

Ciò che queste vulnerabilità hanno rivelato è che lo stato microarchitetturale, invece, non è veramente invisibile.

1.3.1 Meltdown

Ad esempio, per quanto riguarda la cache, abbiamo che si può in qualche modo capire se qualcosa è scritto in cache: bastera invalidare una cacheline, far eseguire il processo che vogliamo studiare, e quindi tentare un'accesso alla stessa cacheline: valutando il tempo necessario all'accesso (e magari facendo una media statistica) si potrà capira se quella cacheline è piena oppure no.

Questa era l'approccio adottato dalla vulnerabilità Meltdown.

```
togli_cacheline

mov indirizzo_vietato, %al # qui il programma muore
# da qui in poi si esegue in speculativa
shl $12, %al
mov vettore (%rax), %rbx

# questo si mette nell'handler del segmentation fault
controlla_cacheline
```

L'istruzione illegale `mov indirizzo_vietato, %al` causerà chiaramente l'arresto del programma, ma le due istruzioni successive verranno comunque eseguite dal processore in modalità speculativa, prima che si manifesti il fault.

La `controlla_cacheline`, invece, viene eseguita comunque attraverso una ridefinizione dell'handler di segmentation fault.

A questo punto, si ha quindi che l'indirizzo letto in `%al`, che il processo non sarebbe autorizzato a vedere, viene trasformato in un indice in un vettore, vettore il cui accesso si traduce nel riempimento di una cachline che rileviamo in `controlla_cacheline`.

Iterando questo processo su tutta la memoria kernel si riesce a ricostruirne una copia, e quindi effettivamente leggere tutta la memoria fisica.

Il problema oggi è stato risolto forzando l'invalidazione della cache ad ogni passaggio al kernel.

1.3.2 Spectre

La vulnerabilità Spectre si basa sulla natura stessa del branch prediction: l'utente può sfruttare la struttura del BTB, che abbiamo detto è effettivamente una cache senza controlli sulle collisioni, per "addestrare" il predittore di branch a fare scelte arbitrarie.

In questo modo si potrà in qualche modo "direzionare" l'esecuzione del kernel in luogo dei salti condizionali, e poi usando metodi come Meltdown capire che cosa il kernel ha fatto.

Anche questo problema si può risolvere invalidando una cache in fase di passaggio al kernel, e in particolare invalidando il BTB.