Sistemi Operativi Avanzati

- Versione originale: Matteo Fanfarillo
- Umile formattazione: Simone Festa

Hardware Insights

1. Introduzione

In generale, non è possibile dire che lo stato di un'applicazione sia semplicemente dato dal "puzzle" degli stati dei componenti software sviluppati. Di fatto, quando mandiamo in esercizio tali componenti software, stiamo generando dei cambi di stato al livello hardware che concorrono a determinare qual è lo stato effettivo del nostro sistema. Tra l'altro, alcuni dei cambi di stato al livello hardware possono non essere voluti o specificati dal programmatore. Il fatto che l'esecuzione di moduli software sviluppati a qualsiasi livello impatti sui moduli sottostanti (e quindi sull'hardware), come vedremo, può rappresentare un problema importante per quanto riguarda la sicurezza: talvolta, per ottenere un sistema più sicuro e performante a livello hardware, sarà necessario ristrutturare l'applicazione software.

Ma quali sono le entità che si frappongono tra ciò che viene specificato dal programmatore e ciò che realmente avviene nel sistema?

- Il **compilatore**: il programma compilato è un oggetto molto complesso che può interfacciarsi direttamente con l'hardware; il compilatore può decidere ad esempio di inserire particolari istruzioni macchina secondo uno specifico ordine in modo completamente trasparente al programmatore.
- Le **hardware run-time decisions**: una volta che è stato generato il flusso esatto delle istruzioni macchina da eseguire per mandare in esercizio l'applicazione, l'hardware in realtà può prendere delle decisioni a run-time su come gestire tale flusso. A parità di istruzioni macchina, processori di vendor diversi possono prendere delle decisioni a run-time differenti.
- La **disponibilità** (o l'assenza) **di specifiche features dell'hardware**.

In pratica, si ha una sorta di non-determinismo dell'hardware e, quindi, del software.

Esempio: se implementiamo l'algoritmo del Panificio di Lamport senza l'ausilio di librerie di sistema (e quindi senza l'ausilio di spinlock o semafori), l'algoritmo a un certo punto della sua esecuzione si romperà. Infatti, le macchine moderne (a meno che non siano single core, dove non è assicurata consistenza globale) non garantiscono una visione consistente per tutti i thread di ciò che sta succedendo in memoria. Prima, ad ogni cpu veniva associato un flusso di esecuzione, cosa non vera per i sistemi moderni.

2. Scheduling e parallelismo nell'architettura di Von Newman

L'architettura di calcolatore più semplice a cui siamo stati abituati a pensare è quella di **Von Newman**, ed è caratterizzata da:

- Un'unica CPU.

- Un’unica memoria.
- Un unico flusso di controllo (*fetch – execute – store*).
- Transizioni nell’hardware time-separated: le istruzioni devono essere eseguite tutte una alla volta.
- Stato della memoria ben definito all’inizio di ciascuna istruzione, la quale quindi deve poter vedere la memoria in uno stato coerente con l’esecuzione delle istruzioni precedenti.

La maniera moderna di pensare le architetture non è basata sull’idea di seguire il flusso di *esecuzione esattamente così com’è* stato codificato nel programma, bensì è basata sul concetto di **scheduling** (e.g. dell’utilizzo dei componenti hardware) che, alla fine della fiera, porterà a eseguire un flusso di esecuzione equivalente al flusso codificato nel programma (gli stati intermedi possono essere diversi, possono cambiare di run in run, non sono deterministici, ma influenzabili da fattori esterni come la temperatura). In particolare, lo scheduling dovrebbe consentire di eseguire più cose in **parallelo**, anche in contesti con un’unica CPU, un’unica memoria e un unico flusso di controllo. Per quanto riguarda lo scheduling, ne esistono diverse tipologie. A livello **hardware** abbiamo:

- Scheduling delle istruzioni all’interno di un singolo program flow.
- Scheduling delle istruzioni in program flow paralleli (**speculativi =** non so se l’esecuzione sia corretta o meno, ad esempio la branch condition non sempre lo è. Il processore sceglie come propagare l’update, non è imposto. Ho garantita l’equivalenza software, non ciò che avviene dentro).
- Propagazione dei valori tra i componenti hardware del sistema.

A livello software invece abbiamo:

- Scheduling dei thread da assegnare alle CPU / ai CPU-core.
- Scheduling delle attività da eseguire sull’hardware (e.g. gli interrupt).
- Supporti di sincronizzazione tra thread software-based.

Anche per quanto riguarda il **parallelismo**, ne esistono diverse tipologie:

- A livello hardware si parla di **ILP** (**Instruction Level Parallelism**), che consiste nell’impiegare le risorse hardware in modo tale da eseguire contemporaneamente istruzioni macchina diverse. (ad esempio posso eseguire al tempo ‘t’ sia A sia B, anche in un unico flusso).
- A livello software, invece, abbiamo il **TLP (Thread Level Parallelism)**, secondo cui un programma può essere pensato come la combinazione di molteplici flussi di esecuzione concorrenti. (Ad esempio ho 3 flussi su 3 processori,singolarmente sarebbero ILP, che cambiano).

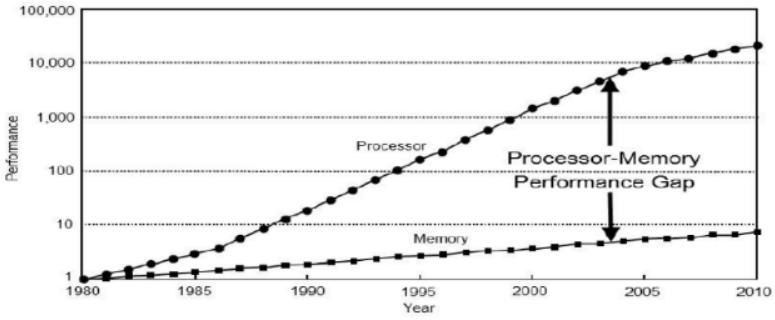
3. Velocità di computazione

È generalmente correlata alla velocità di un processore (espressa in GHz), anche se in realtà esistono istruzioni che possono richiedere un numero arbitrario di cicli di clock a causa di più possibili fattori:

- Possono essere istruzioni più onerose per loro natura.

- Possono dover richiedere a un certo punto una risorsa hardware tuttora occupata da un'altra istruzione, per cui devono rimanere in attesa.
- Possono esservi delle asimmetrie a livello hardware (e.g. un CPU-core può essere più veloce di un altro).
- I pattern per l'accesso ai dati influiscono a loro volta sulle prestazioni: ad esempio, se un dato viene memorizzato in cache, l'accesso a esso sarà più efficiente e viceversa.

Nel corso base di Sistemi Operativi abbiamo parlato di thread CPU-bound e di thread I/O-bound; introduciamo ora una terza categoria di thread (che, di fatto, è una sottocategoria dei CPU-bound): i **memory-bound**. Essi sono dei thread che utilizzano in maniera intensiva la CPU ma, mentre sono in esecuzione, utilizzano in maniera intensiva anche la memoria. I thread (o comunque i programmi) che presentano questa caratteristica possono rappresentare un problema dal punto di vista prestazionale: come riportato dal seguente grafico, il divario prestazionale tra processore e memoria aumenta sempre di più col tempo; questo fenomeno è detto **memory wall**.



Per evitare che i thread memory-bound sperimentino e causino ad altri thread un crollo delle prestazioni, sono necessari dei meccanismi avanzati ad-hoc al livello dell'hardware.

4. Pipeline

È una **tecnica di scheduling e parallelismo hardware-based**. Infatti:

- Non prevede una separazione temporale tra le finestre di esecuzione delle diverse istruzioni: più istruzioni possono essere in esecuzione contemporaneamente (questo è *parallelismo*).
- Le istruzioni che vengono sequenzializzate dal programmatore non sono necessariamente eseguite secondo la stessa sequenza nell'hardware, anche per conseguire la proprietà di parallelismo (questo è *scheduling*).

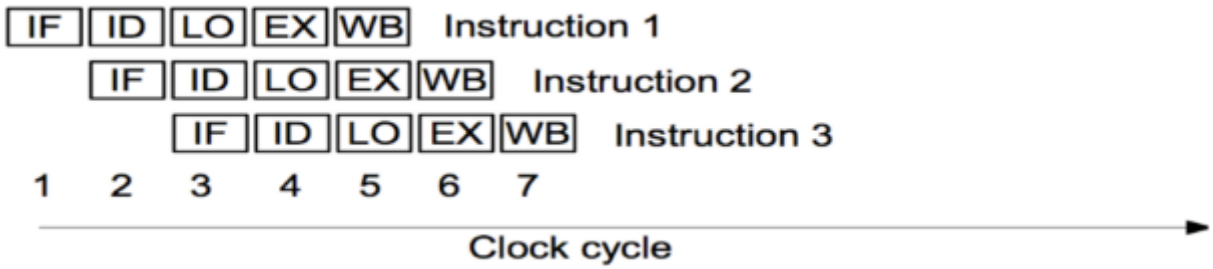
In ogni caso, ci sono spesso e volentieri coppie di istruzioni (*i1*, *i2*) in cui *i2*, per essere eseguita, necessita del risultato ottenuto da *i1*. In tal caso, *la causalità deve essere preservata*, per cui *i1* e *i2* non possono essere schedulate in modo del tutto arbitrario. Questo non è altro che un modello **data flow** per l'esecuzione dei programmi.

Ricordiamo che le fasi (stage) delle istruzioni sono:

- **IF** (**Instruction Fetch**): caricamento dell'istruzione nel processore (richiede certamente un accesso in memoria).
- **ID** (**Instruction Decode**): decodifica dell'istruzione, in cui viene stabilito ciò che deve effettivamente essere fatto per eseguire l'istruzione stessa.

- **LO** (**Load Operands**): caricamento degli operandi richiesti per l'esecuzione dell'istruzione (potrebbe richiedere un accesso in memoria).
- **EX** (**Execute**): esecuzione vera e propria dell'istruzione.
- **WB** (**Write Back**): scrittura dell'output dell'istruzione su un registro o in memoria (potrebbe quindi richiedere un accesso in memoria).

Il fatto che un thread o un'applicazione sia memory-bound o meno dipende proprio dagli stage *LO* e *WB*. Inoltre, per permettere l'esecuzione di più istruzioni in parallelo, è necessario che ciascuno *stage coinvolga una componente hardware differente del processore*, in modo tale da non avere collisioni tra più istruzioni che vengono eseguite contemporaneamente. In particolare, un'astrazione di base della pipeline è quella riportata nella seguente figura.



In questo scenario, idealmente, sarebbe possibile completare un'istruzione per ogni ciclo di clock (anche se poi nella realtà non è esattamente così per i motivi esposti nel paragrafo "Velocità di computazione"). Supponiamo comunque di trovarci nello scenario ideale in cui ciascuno stage viene eseguito in un unico ciclo di clock, e supponiamo di voler fornire N risultati (1 per ogni istruzione), di avere L diversi stage per le istruzioni (gli stage sarebbero *IF*, *ID*, *LO*, *EX* e *WB* che devo attraversare) e di avere un ciclo di clock di durata pari a T.

- Senza pipeline si ha un ritardo pari a $N \cdot L \cdot T$
- Con la pipeline si ha un ritardo pari a $(N + L) \cdot T$

Lo speedup è dunque pari a $(N \cdot L)/(N + L)$, che tende a L per N tendente a ∞. Dal punto di vista delle prestazioni, sarebbe magnifico avere un L molto grande (ovvero molti stage diversi per le istruzioni, ovvero riesco a parallelizzare molto di più se aumento il numero di componenti parallelizzabili) ma ciò nella realtà non accade. Il caso estremo richiederebbe avere hardware infinito, visto che ogni stage è un pezzo hw! Nella realtà:

- I processori Pentium prevedevano 5 stage.
- I processori i3 / i5 / i7 prevedono 14 stage.
- I processori ARM-11 prevedono 8 stage.

Questa scelta è dovuta alla necessità di preservare la causalità tra le istruzioni. Più istruzioni si prendono in considerazione insieme, più è probabile che tra tali istruzioni ce ne siano alcune (e.g. *i1, i2*) legate da una relazione di causalità, e sappiamo che *i2*, per essere eseguita, deve attendere che il risultato di *i1* sia pronto; in tal caso, più L è grande, più la pipeline è lunga, più l'attesa di *i2* sarà lunga. Lo scenario di cui abbiamo appena parlato è una **data dependency**. Oltre a questa esiste anche la **control dependency**, che si può avere nel momento in cui c'è un salto condizionale il cui esito dipende dagli outcome delle

istruzioni precedenti al salto. Per quanto concerne la data dependency tra le istruzioni *i1* e *i2*, abbiamo che lo stage **LO** di *i2* tipicamente non può precedere lo stage **WB** di *i1*. Analogamente, per quanto riguarda la control dependency tra le istruzioni *i1*e *i2* (dove *i1* è l'istruzione di salto condizionato), non si conosce l'esito del salto prima della fase **EX*** di *i1*, per cui la fetch di *i2* non dovrebbe precedere lo stage **EX*** di *i1*. Tutte queste condizioni possono comportare dei rallentamenti nell'esecuzione dell'applicazione rispetto al caso ideale di pipeline. Per gestire tali condizioni è possibile ricorrere a svariati meccanismi:

- **Stalli software** (compiler driven): possono essere aggiunti all'interno della pipeline con lo scopo di distanziare due istruzioni *i1*, *i2* in modo tale che uno stage x (e.g. LO) di *i2* venga eseguito dopo uno stage y (e.g. WB) di *i1*.

- **Rischedulazione software** (compiler driven): se devono essere eseguite tre istruzioni *i1*, *i2*, *i3* tali per cui *i2* dipende da *i1* ma *i3* non dipende né da *i1* né da *i2*, il compilatore può frapporre *i3* tra *i1* e *i2* in modo tale da svolgere lavoro utile mentre *i2* è in attesa che si completi uno specifico stage di i1.

- **Propagazione hardware**: se l'istruzione *i2* dipende dall'istruzione *i1* e, ad esempio, lo stage LO di *i2* consiste nell'acquisire un valore prodotto dallo stage EX di *i1*, allora è possibile per *i1* propagare il valore a *i2* subito dopo la fase EX senza dover attendere il completamento della write-back.

- **Azzardi hardware supported**: contestualmente a un'istruzione di salto condizionale, il processore può provare a indovinare se il salto viene preso o meno per poi anticipare la fetch delle istruzioni successive di conseguenza. Dal punto di vista delle performance, una predizione errata equivale a un inserimento di stalli per attendere passivamente l'esito dell'istruzione di salto; di conseguenza, la tecnica degli azzardi è statisticamente conveniente da adottare.

- **Rischedulazione hardware**: è un meccanismo noto anche come **out-of-order pipeline** (**OOO**), e prevede che le istruzioni vengano completate non necessariamente nel medesimo ordine con cui sono entrate all'interno della pipeline. In particolare, un'istruzione i_k, per accedere allo stage x, non deve necessariamente attendere che tutte le istruzioni a lei precedenti abbiano completato lo stage x. Si può dunque avere un meccanismo di **superamento** delle istruzioni all'interno della pipeline. Tale tecnica è vantaggiosa poiché permette all'istruzione i_k di essere completata prima e, quindi, di liberare un posto all'interno della pipeline. Tuttavia, è una tecnica che *richiede la possibilità da parte dell'hardware di ospitare più istruzioni contemporaneamente all'interno dello stesso stage* (altrimenti non sarebbe possibile effettuare il sorpasso): i processori con questa caratteristica sono detti **superscalari**. Inoltre, ovviamente, affinché si possa avere una out-of-order pipeline, *l'istruzione che effettua il sorpasso non deve dipendere dalle istruzioni che vengono sorpassate.* Non si hanno effetti reali sull'**Instruction Set Architecture** finchè non si deve "mostrare" l'output.

Esempio:

Se in pipeline su un thread ho $A \rightarrow B$, e nella pipeline B supera A, ma A è oggetto di trap (quindi non può fare quello che stava facendo, come dividere per 0, di conseguenza non potrò averla in ISA), cosa accade a B? Vedremo che in OOO si producono valori registrati in maniera "speculativa" che poi butterò, ma non posso eseguire una UNDO.

Nota: le istruzioni tra due thread sono indipendenti, dipendono solo se usano info condivise, ma ciò è di interesse al programmatore, non al processore. L'ordine della sorgente (programma) non è detto coincida con l'ordine del compilatore, tuttavia abbiamo la sicurezza che il data flow sia compatibile. A livello

hardware, se ho operazione x, y, z può capitare quindi di avere z, x, y; ma ciò è realizzato dinamicamente dall'hardware. Ho sorpasso se non ho dipendenza.

4.1. Pipeline vs Sviluppo

Come abbiamo visto, i programmatori non hanno il diretto controllo del comportamento di un processore (trattasi di microcodice) ma, comunque sia, il modo con cui viene scritto il software può impattare sulle prestazioni effettive della pipeline. A livello ISA vedo il set di istruzioni e risorse usabili, ma non vedo la pipeline. Esempio: Esempi di ISA sono *ADD, COMPARE, JUMP*.onsideriamo le seguenti istruzioni C:

1. `a = *++p`
2. `a = *p++`

L'istruzione 1 prevede che prima debba essere incrementato il puntatore p affinché referenzi la entry successiva e solo dopo si possa accedere alla entry appena referenziata; di conseguenza, l'accesso dipende dall'incremento del puntatore. Al contrario, l'istruzione 2 prevede che prima si debba accedere alla entry attualmente referenziata dal puntatore p e solo poi si debba incrementare p; stavolta, l'accesso non dipende dall'incremento del puntatore. Questo vuol dire che c'è dipendenza. Consideriamo due programmi, di cui il primo prevede l'istruzione 1 all'interno di un loop e il secondo prevede l'istruzione 2 all'interno di un loop. Nel secondo c'è indipendenza. La differenza prestazionale tra i due programmi è abbastanza significativa (può raggiungere tranquillamente il $20/25\%$).

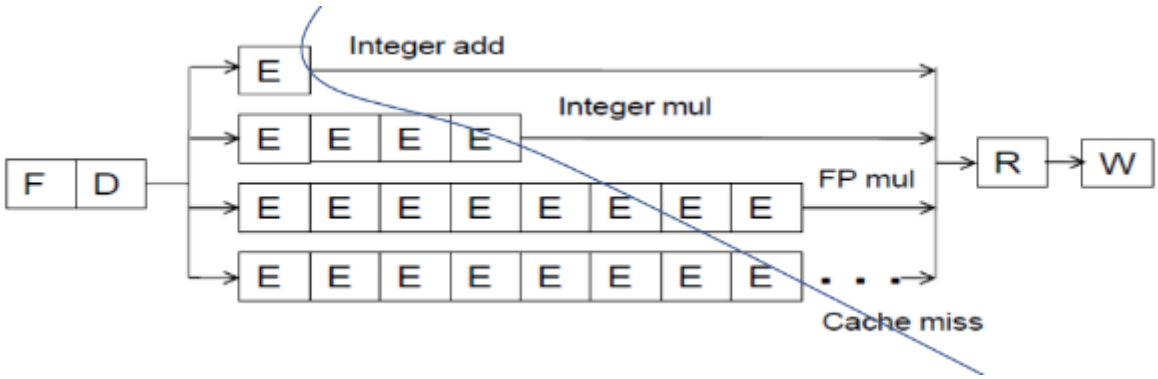
Per giunta, esistono delle istruzioni macchina che, da sole, hanno degli effetti devastanti sulle prestazioni del programma. Un esempio è `cpuid`, che ha lo scopo di restituire l'id del processore (o del CPU-core o dell'hyperthread) che ha processato l'istruzione stessa. Ma, oltre a questo, effettua anche lo **squash** della pipeline: in particolare, nel momento in cui `cpuid` viene realmente eseguita, la pipeline viene svuotata e le altre istruzioni al suo interno vengono buttate. Questo non deve necessariamente rappresentare uno svantaggio: avere istruzioni pendenti all'interno della pipeline significa dire che tali istruzioni possono essere schedulate dinamicamente dall'hardware secondo regole non meglio identificate, e ciò può impattare non solo sulla correttezza, ma anche sulla sicurezza del sistema. Più precisamente, quello di flushare la pipeline è un concetto legato al termine "**serializzazione**". Di fatto, `cpuid` è detta **istruzione serializzante**, poiché riporta la pipeline a lavorare secondo uno schema sequenziale. Non solo: `cpuid`, come tutte le istruzioni serializzanti, garantisce che qualunque modifica apportata a flag, registri e memoria da parte delle istruzioni precedenti sia completata (finalizzata) prima che una qualsiasi istruzione a lei successiva venga fetchata ed eseguita. *Ciò implica anche che cpuid non può superare alcuna istruzione davanti a lei nella pipeline.* (*e.g:* Perchè le istruzioni successive devono ancora essere fetchate ed eseguite, quindi come potrei superarle? Ciò che viene dopo questa istruzione serializzante è come se andasse in stallo finchè non completo questa istruzione serializzante, e garantisce anche che le istruzioni precedenti vengano viste da quelle successive.)
Se avessi SOLO istruzioni serializzanti, il sistema sarebbe molto più lento.

4.2. Pipeline superscalare

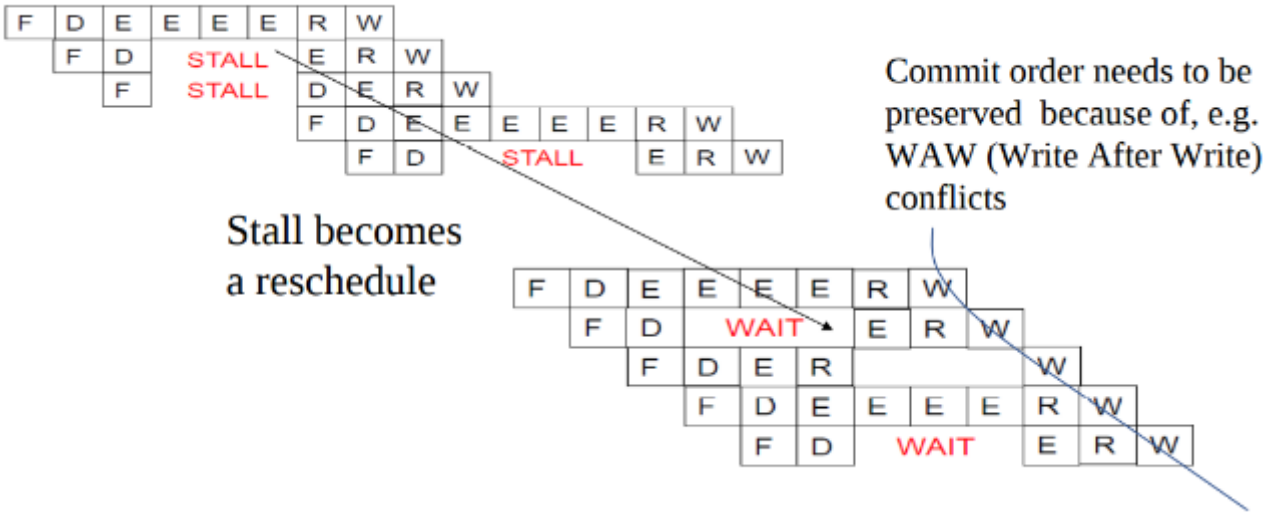
È un insieme di molteplici pipeline che operano simultaneamente all'interno del processore. La si può ottenere aggiungendo *ridondanza alle risorse hardware* in modo tale che più componenti distinti siano adibite a uno stesso stage della pipeline. Come detto in precedenza, questa possibilità permette anche di adottare il modello OOO, la cui idea di base consiste in:

- Effettuare il **commit** (o **retire** o **finalizzazione**) delle istruzioni esattamente nel medesimo ordine in cui sono *entrate nella pipeline*, indipendentemente dagli eventuali sorpassi avvenuti all'interno della pipeline. Ciò significa che le scritture dei risultati delle istruzioni in memoria, su un registro qualunque o su un registro di stato *devono avvenire esattamente nell'ordine prestabilito*.
- **Processare le istruzioni independenti** (sia sui dati che sulle risorse hardware) **il prima possibile**, dove un'istruzione indipendente sulle risorse hardware è un'istruzione che non ha bisogno di attendere che un qualche componente si liberi per processare un certo stage x.

L'immagine riportata di seguito mostra molto bene come lo stage EX delle istruzioni possa avere una durata variabile in termini di numero di cicli di clock, ed è a partire da tale presupposto che risulta utile avere un out-of-order pipeline.



La seguente altra figura mostra invece la differenza prestazionale che si può avere tra il modello non OOO e il modello OOO:



Osservazione: Con una pipeline che ha ridondanza hardware (come i pc moderni), possiamo ad esempio parlare di "*Pipeline parallele a X canali*", che è come avere una autostrada a X corsie. Con Out-Of-Order Pipeline si evitano "grosse latenze", poichè se "libero una corsia perchè sono veloce" anche gli altri andranno più veloci (liberarando risorse utili ad altri). Lo stallo impatta su tutti invece. Nella O.O.O, se posso andare avanti, lo faccio, non mi importa che devo aspettare il commit, se posso eseguire qualcosa lo eseguo. Per questo parliamo di WAIT e non più di STALL. Nella WAIT, appena ho un risultato (outcome di una "corsia") lo fornisco a chi lo necessita, non mi serve aspettare che si liberi tutta la corsia.

Ora, poiché tra l'**emission** (= iniezione all'interno della pipeline, normalmente di più di una istruzione) e il retire (= **commit**) delle istruzioni si ha una fase di esecuzione in cui le istruzioni stesse possono sorpassarsi a vicenda, si va incontro al cosiddetto problema delle **eccezioni imprecise (OFFENDING)**: supponiamo di avere un'istruzione *i2* che ha superato un'istruzione *i1*, e assumiamo che *i1*, durante lo stage EX, generi un'eccezione (*e.g*: perché magari si tratta di una divisione per 0); a questo punto, anche se il risultato di *i2* non è stato ancora committato e, quindi, esposto a livello di ISA, *i2* può aver comunque toccato delle risorse non esposte a livello di ISA ed effettuato dunque dei cambi di stato (in particolare cambi di **stato micro-architetturale**). Quindi con offending instructions non esponiamo su ISA, ma potrei cambiare lo stato hardware, il che è usabile da malintenzionati. Questo perchè, lavorare in un contesto Out-Of-Order permette il superamento di istruzioni (**SPECULAZIONE**) che lascia delle tracce. Di conseguenza, l'eccezione sollevata da *i1* vede uno stato interno dell'hardware che ingloba anche delle attività relative a *i2*. Un problema analogo lo si ha anche a parti invertite: se l'istruzione che genera l'eccezione è *i2*, l'eccezione vedrà uno stato interno dell'hardware che non comprende il risultato prodotto da *i1*. Questo è un problema dello stato e delle informazioni all'interno del processore (*ma non esposte nell'ISA*). Peggio ancora: *i2* potrebbe sollevare un'eccezione che in realtà, secondo il program flow, non sarebbe dovuta mai esistere, magari perché *i1* è a sua volta un'istruzione che solleva un'eccezione o un'istruzione di salto. In realtà, si possono avere eccezioni imprecise anche in assenza di sorpassi: se l'istruzione *i2* viene dopo l'istruzione *i1* che genera un'eccezione, *i2* può aver comunque attraversato degli stage e, quindi, può aver modificato lo stato interno dell'hardware. Come vedremo, tutto questo rappresenta un grave problema per la sicurezza dei sistemi: **Meltdown** è solo il primo di una valanga di attacchi che hanno sfruttato tale vulnerabilità.

5. Algoritmo di Robert Tomasulo

Secondo Robert Tomasulo, in uno scenario di utilizzo di una pipeline speculativa out-of-order (dove speculativa = caratterizzata dall'esecuzione di istruzioni che potrebbero servire solo in un secondo momento), se consideriamo due istruzioni A, B tali che $A \rightarrow B$ nell'ordine di programma, dobbiamo stare attenti nell'evitare i seguenti tre tipi di azzardo:

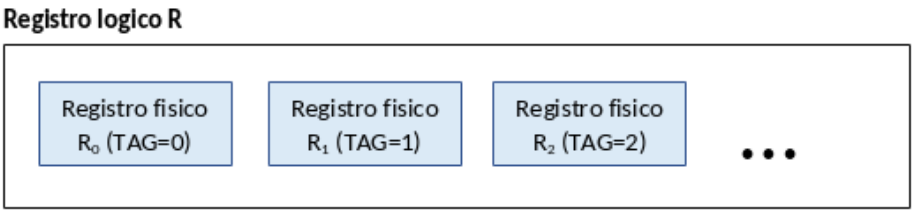
- **RAW** (**Read After Write**): B deve leggere un dato R necessariamente dopo che A lo ha aggiornato.
- **WAW** (**Write After Write**): B deve scrivere su un dato R necessariamente dopo che A lo ha aggiornato.
- **WAR** (**Write After Read**): B deve scrivere su un dato R necessariamente dopo che A ne ha letto il valore precedente (altrimenti vorrebbe dire che A è in grado di leggere dal futuro).

5.1. Dipendenza RAW

Qui è necessario bloccare l'istruzione B e tenere traccia di quando il dato che deve essere letto da B sarà disponibile (disponibile non vuole dire necessariamente committato).

5.2. Dipendenza WAW e WAR

Qui è necessario adottare una tecnica nota come **register renaming**, secondo cui al programmatore vengono esposti dei registri logici, e ciascuno di questi registri logici (che sono di fatto dei multi-registri) ingloba un insieme di registri fisici non visibili al livello dell'ISA. Quindi noi non scriviamo MAI sul registro esposto in ISA, ma su delle 'copie numerate' o alias, cioè registri multivalore. Ci ritroviamo dunque nella seguente situazione:



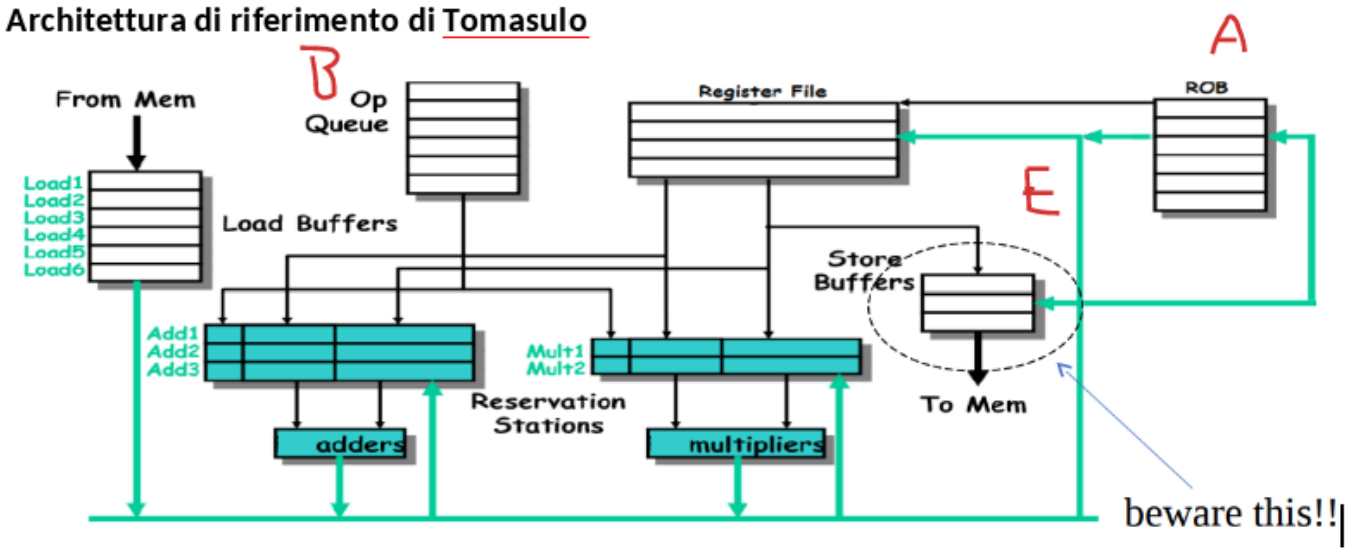
Qui i vari registri fisici rappresentano diverse versioni del medesimo registro logico *R*. Nel momento in cui all'interno della pipeline entra un'istruzione che vuole scrivere sul registro logico *R*, si considera il primo registro fisico R_k all'interno del quale viene effettivamente memorizzato il valore (quindi scrivo sempre sul primo tag libero, e leggo dall'ultimo tag in pipeline. Se A scrive in TAG 0, e B scrive in TAG 1, leggo da TAG1). Tipicamente, per la selezione del registro fisico, si segue un approccio round-robin; se però a un certo punto tutti i registri fisici di *R* sono stati sovrascritti e nessuna delle istruzioni che ha eseguito la scrittura è andata in commit, per un'eventuale altra scrittura su *R* bisognerà attendere.

Posso superare solo dopo essere entrato nella pipeline (nelle condizioni discusse nelle pagine precedenti), ma non posso superare prima di entrare in pipeline.

In pratica, un *renamed register* materializza il concetto di speculatività di una pipeline: quando vengono effettuate delle write, vengono scritti dei valori che non necessariamente verranno considerati come validi, ma che comunque potranno essere letti dalle istruzioni successive (che sono state introdotte nella pipeline speculativamente).

Relativamente al registro *R*, vengono memorizzati anche dei metadati di gestione di *R* che indicano qual è il tag contenente la versione committata; la versione committata è la versione del valore di *R* scritto dall'ultima istruzione andata in commit che ha toccato proprio *R*. In questi registri abbiamo il "futuro" che avverrà, perchè non li ho ancora committati.

6. Architettura di riferimento di Tomasulo



- **A)**: I metadati associati alle istruzioni fetchate vengono memorizzati all'interno del **ROB** (**Re-Order Buffer**), che ci aiuta a scrivere le istruzioni nello store buffer, in quanto posso andare in memoria solo se sono committed, ma io non posso aspettare sempre questa fase prima di utilizzare i dati. Tali metadati possono essere:
    - *L'ordine con cui le istruzioni dovranno andare in commit.*

    - Che cosa dovrebbero fare le istruzioni nella loro esecuzione speculativa.
    - Qual è l'alias (l'istanza fisica) dei registri che dovranno essere usati da ciascuna istruzione; ad esempio, se un'istruzione A deve scrivere su un certo registro R e un'istruzione B successiva deve leggere dallo stesso registro R con una dipendenza RAW, è chiaro che A e B dovranno utilizzare il medesimo alias.

- **B)**: Le operazioni vere e proprie da eseguire (quindi gli OP code delle istruzioni) vengono memorizzate all'interno dell'**OP queue** e, in base alla loro tipologia, verranno date in input a un particolare componente di processamento (add, mult e così via). Nel momento in cui un'istruzione è pronta per essere processata, devono essere recuperati i metadati associati a essa e, in particolare, gli alias dei registri da usare. A tale scopo, c'è un bus comune (detto **Common Data Bus – CDB**) che mette in comunicazione i componenti di processamento col ROB. Se un alias non è pronto per essere utilizzato (perché magari bisogna attendere che venga sovrascritto da un'istruzione precedente), l'istruzione viene posta in attesa all'interno del relativo componente di processamento. I componenti di processamento sono detti anche **reservation station**. *Cosa è Reservation Station?* Per ogni componente in grado di eseguire uno specifico calcolo, si ha una coda/buffer/corsia dove posso mettere varie istruzioni, e le operazioni identificate dai registri usati (sorgenti). Non si ha sorpasso per utilizzare tali componenti in queste code, al massimo se ho due istruzioni dipendenti cerco di metterle nella stessa reservation station.

- Si fa uso di una **cache** per leggere in modo più efficiente i dati provenienti dalla memoria.

- Nel momento in cui viene **committata** un'istruzione, l'eventuale alias aggiornato da tale istruzione viene installato all'interno del **register file** come valore valido, ovvero come **valore esposto all'interno dell'ISA.**

- **E)**: Se l'istruzione committata prevede una scrittura in memoria, il valore di output, prima di essere riportato in memoria, viene inserito nello **store buffer** in modo tale da velocizzare il completamento dell'istruzione. Tuttavia, ciò implica che il valore non sarà in memoria per un po' di tempo, il che può rappresentare un problema dal punto di vista della consistenza.
  Nota: Supponiamo che un'istruzione sia in fase di ritiro, quindi è totalmente conclusa. In questo istante, il dato dovrebbe passare da Store Buffer alla memoria, ma in realtà non è istantaneo, passa un piccolo lasso di tempo. Ciò crea problemi con > 1 thread, perchè se pesco un dato in memoria potrei non vedere alcuni dati. Per questo l'algoritmo della "*pasticceria*" non funziona nei processori moderni, suppone che tale tempo sia nullo. Esistono però delle istruzioni per controllare lo stato dello Store Buffer. Questo approccio è l'unico per lavorare con O.O.O, ovvero non esistono altre tecniche per affrontare le O.O.O pipeline.

7. Esempi di schemi di esecuzione

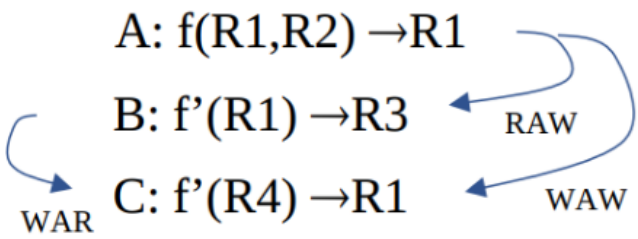
7.1. Esempio 1

Supponiamo di avere tre istruzioni *A, B, C* tra cui *B* e *C* hanno una stessa latenza d, mentre *A* ha una latenza $d' > d$.
Assumiamo inoltre che:

- *A* esegua l'operazione $f(R1, R2)$ e riporti l'output sul registro *R1*. (in particolare, scrive su un alias di *R1*).

- *B* esegua l'operazione $f'(R1)$ e riporti l'output sul registro *R3*. (legge da STESSO alias di R1)
- *C* esegua l'operazione $f'(R4)$ e riporti l'output sul registro *R1*. (scrive su UN ALTRO alias di R1).

Ci ritroviamo dunque nel seguente scenario:



È abbastanza evidente che *B* debba leggere **lo stesso alias** scritto da A, mentre C potrà riportare il valore di output su un **alias differente**. In tal modo è possibile **processare** C **parallelamente ad** A :
In fondo, anche se *C* genera il suo output prima di *A*, le dipendenze che legano le tre istruzioni non vengono violate. Infatti, *B* sarà comunque in grado di leggere dall'alias sovrascritto da *A* e, ovviamente, rimarrà comunque possibile mandare in commit *C* solo dopo il completamento di *A* e *B*. Il vantaggio che porta questo approccio è la riduzione del tempo necessario per il completamento di *C*.

7.2. Esempio 2

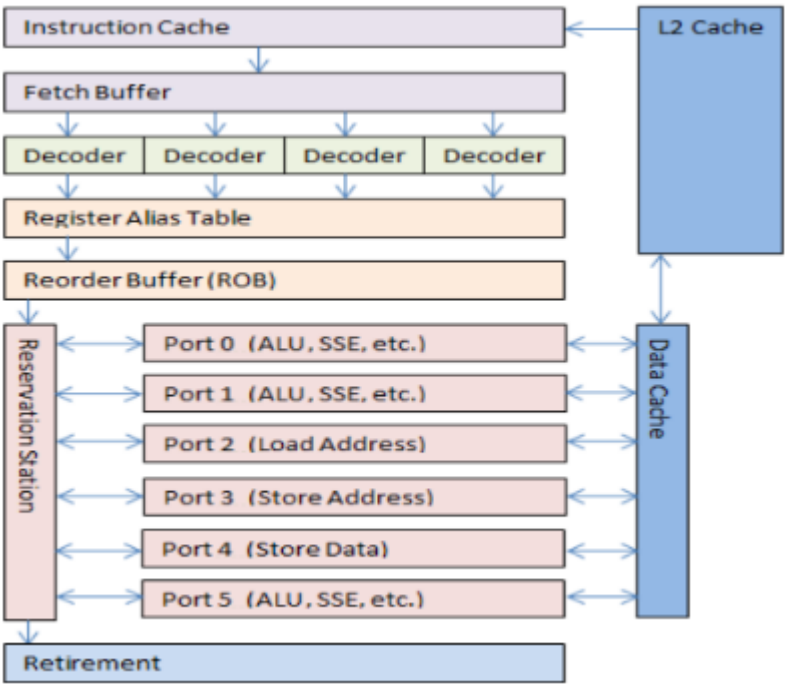
Vediamo con un secondo esempio riportato qui di seguito come viene associata una entry del ROB a ciascun alias differente:

Before:		after	
	add r3,r3,4		add rob6,r3,4
	add r4,r7,r3		add rob7,r7,rob6
	add r3, r2, r7		add rob8,r2,r7

E' importante ricordare che, anche se è possibile eseguire in parallelo, non posso pensare di aumentare all'infinito le prestazioni, perchè ad un certo punto eccedo il numero di istruzioni che posso gestire. Questo porta ad un sistema scarico, ovvero non occupo corsie perchè ci sono caricamenti di elementi dalla memoria, oppure eseguo una predizione/azzardo sbagliata che mi porta a svuotare etc...
E' come fare un'autostrada a 20 corsie: è molto difficile saturarla il "giusto".

Che soluzione è possibile adottare? Introduciamo i **Processori HYPERTHREAD**.

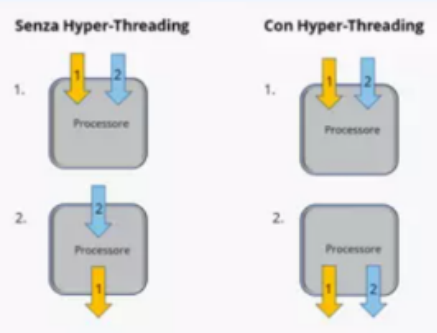
8. Organizzazione architetturale dei processori x86 OOO



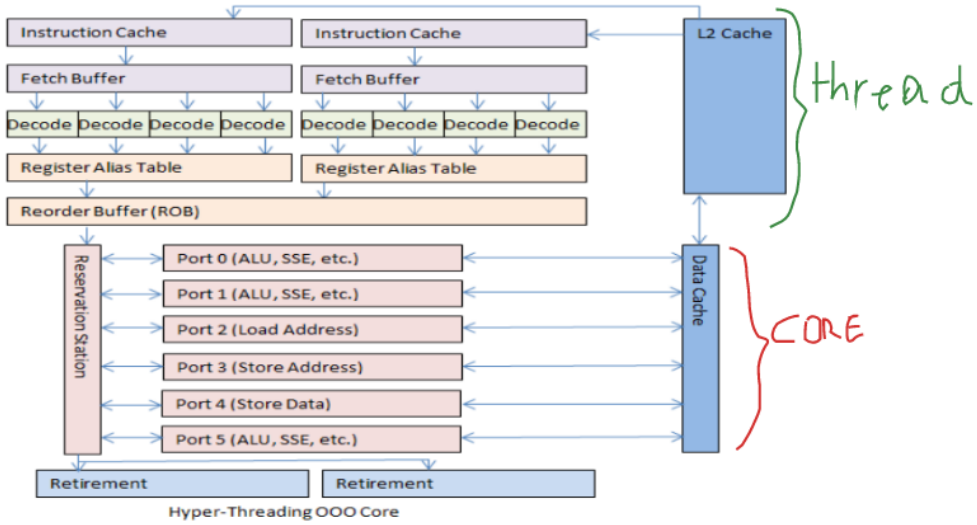
8.1. Processori hyper-threaded

Come abbiamo osservato all'inizio della trattazione, le architetture moderne sono soggette al **memory wall**. In particolare, si ha una latenza di esecuzione non solo quando si devono *recuperare i dati dalla memoria*, ma anche quando devono essere estratte le istruzioni stesse. Dunque, si può anche avere una pipeline molto efficiente con un **ILP** elevato, ma il processore non arriva mai a processare in parallelo tutte le istruzioni possibili a *causa dei ritardi causati dalla memoria*, per cui la potenza e le risorse del processore risultano sprecate. Per ovviare all'inconveniente, si può assegnare una *stessa reservation station a più program flow diversi*. Da qui nascono i processori **hyper-threaded**, che hanno un *unico core fisico (i.e. un'unica information station)* che permette di eseguire le reali operazioni dettate dalle istruzioni; d'altra parte, la porzione di processore che memorizza le istruzioni da fetchare, decodifica le istruzioni per uno specifico flusso (Decode) e tiene traccia dei registri logici dell'ISA (mediante Register Alias Table) viene replicata e viene definita **hyperthread**. Una CPU può contenere uno o più core. Un core è un'unità all'interno della CPU che realizza l'effettiva esecuzione. E' un "oggetto engine". Con l'Hyper-Threading, un microprocessore fisico si comporta come se avesse due core logici, ma virtuali. In questo modo si consente a un unico processore l'esecuzione di più thread contemporaneamente. Questo procedimento aumenta le prestazioni della CPU e migliora l'utilizzo del computer.

Esempio: ho una bocca (core) e due mani con cui prendo il cibo (thread). Per migliorare la prestazione devo puntare ad aumentare i thread (più mani = più cibo in entrata). Se aumentassi il numero di bocche, non migliorerei, perchè sempre quel cibo ingerisco, anche se finisco prima devo aspettare la prossima forchettata.



In tal modo, è possibile eseguire più **workflow in parallelo su un unico core fisico**, e l'architettura risultante è riportata nella pagina seguente:



Tale architettura risulta molto vantaggiosa per la maggior parte dei workload, mentre risulta un po' più avversa nel caso in cui tutti i thread che girano sullo **stesso core sono CPU-bound** ma non memory-bound. Infatti, in quest'ultimo caso è più probabile che si verifichi un conflitto tra i thread in esecuzione nell'utilizzo delle risorse della reservation station. Un processore hyper-threaded viene tipicamente marcato con l'indicazione su quanti core (="motori") e quanti hyperthread (="thread fisici") possiede.
Esempio:
Un processore a due core e quattro thread fisici viene marcato come **2C/4T**.
In un contesto senza Hyperthread avremmo 2 core e 2 thread (1 per ogni core).
In un contesto con Hyperthread possiamo avere 2 core e 4 thread (2 thread per ogni core).
Se ho un caso 2C/4T, vuol dire che il mio programma non può generare più di 4 thread?
No, vuol dire che ogni core può gestire al massimo due thread con un proprio set di registri aventi alias (quindi due workflow in parallelo su unico core), quindi in totale posso gestire al massimo quattro thread/workflow in contemporanea. Ci sarà lo scheduling che realizzerà il parallelismo, ma comunque ne verranno gestiti quattro insieme. I flussi sui thread possono essere speculativi.

9. Gestione degli interrupt

Ne sono esempi: muovere mouse, premere sulla tastiera.
L'interrupt è un segnale *proveniente da un certo dispositivo hardware* che indica la necessità di **cambiare il flusso di esecuzione** (e.g. andando a eseguire un handler). Poiché è buona norma processare gli interrupt il prima possibile, quando ne occorre uno, si attende solo che una delle istruzioni correntemente in pipeline vada in commit; dopodiché si effettua lo squash (ovvero si svuota) la pipeline e si iniziano a fetchare le istruzioni dell'handler dell'interrupt. Ciò può portare a un rallentamento dell'esecuzione, ma non vale la pena memorizzare da qualche parte lo stato di esecuzione delle istruzioni che vengono buttate (richiederebbe hardware aggiuntivo, inoltre non ho certezza che dopo l'handler, ritornando al

flusso di esecuzione, io parta dall'istruzione subito dopo); tra l'altro, anche mentre viene eseguito il gestore di un certo interrupt può subentrare un ulteriore interrupt, e questo può avvenire iterativamente un numero arbitrario di volte.

Attenzione:
Buttare delle istruzioni dalla pipeline implica prevenire i loro effetti sull'ISA ma, se esse hanno sporcato lo stato micro-architetturale, quest'ultimo non può essere ripristinato: rimangono comunque gli effetti dell'esecuzione speculativa delle istruzioni che sono state buttate. A livello hardware non ho meccanismi di gestione priorità o trap, operazioni offending come una divisione per 0 non vengono svolte dal processore, ma passano ad un handler.

10. Gestione delle eccezioni

Esse avvengono durante esecuzione di un programma, a livello software. Le eccezioni risultano più complesse da gestire rispetto agli interrupt poiché vengono sollevate dall'esecuzione di particolari istruzioni. Di fatto, un'istruzione A (cosiddetta **offending**) che solleva un'eccezione e_A può essere eseguita speculativamente all'interno della pipeline e, di conseguenza, potrebbe non esistere nel flusso di esecuzione definito dal programmatore. Ad esempio, poco prima dell'istruzione A potrebbe esserci un'istruzione B che solleva a sua volta un'eccezione e_B: in tal caso sarà e_B a esistere realmente e non e_A. Una Istruzione è offending se vuole usare qualcosa che non è usabile. Non genera una trap, perchè non so se arrivo in retire. Se l'istruzione successiva in fondo non viene committata, cancello tutto. A valle di queste considerazioni, un'*eccezione viene presa in carico solo nel momento in cui l'istruzione che l'ha generata va in retire*, anche se ci si può accorgere prima che l'istruzione sia offending. (Vedi sotto, etichettamento offending).
Ma quali sono gli stage in cui un'istruzione può rivelarsi offending?

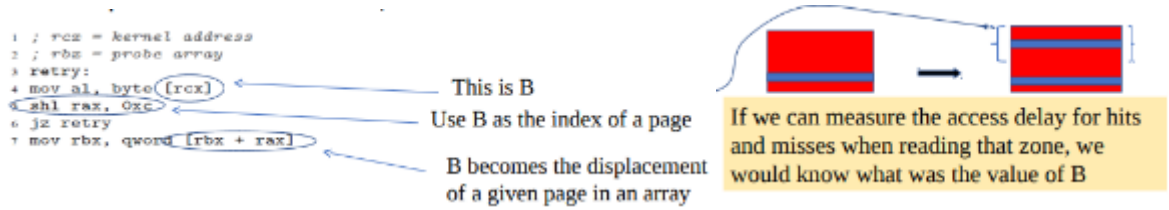
- **Instruction Fetch / Memory stages**
  (MEM stage = stage in cui avviene l'accesso agli operandi): qui si può avere un page fault (ovvero un tentato accesso a un indirizzo logico di memoria che attualmente non ha un corrispettivo fisico), un accesso in memoria disallineato o una violazione delle protezioni applicate a una pagina di memoria (e.g. si tenta di accedere in scrittura a una locazione read-only).
- **Instruction Decode stage**: qui può emergere che l'istruzione in esercizio sia illegale.
- **Execution stage**: qui può essere sollevata un'eccezione aritmetica (e.g. divisione per zero).
- **Write-Back stage**: qui non possono essere sollevate eccezioni.

Per etichettare un'istruzione come offending, si imposta a 1 un apposito bit dei metadati. Quando tale istruzione va in *retire*, se il bit vale 1 allora il commit non viene effettuato, bensì viene attivato il gestore di eccezioni opportuno. A tal punto tutte le istruzioni successive a quella offending diventano *phantom* (fantasma).

Attenzione: Con le eccezioni si verifica lo stesso problema riscontrato con gli interrupt. In particolare, quando un'istruzione offending va in retire, si hanno altre istruzioni all'interno della pipeline che hanno già modificato lo stato micro-architetturale e, dunque, hanno lasciato tracce di informazione. Questa problematica lascia spazio al cosiddetto attacco **Meltdown**. Sto propagando "attività illecite" nella pipeline.

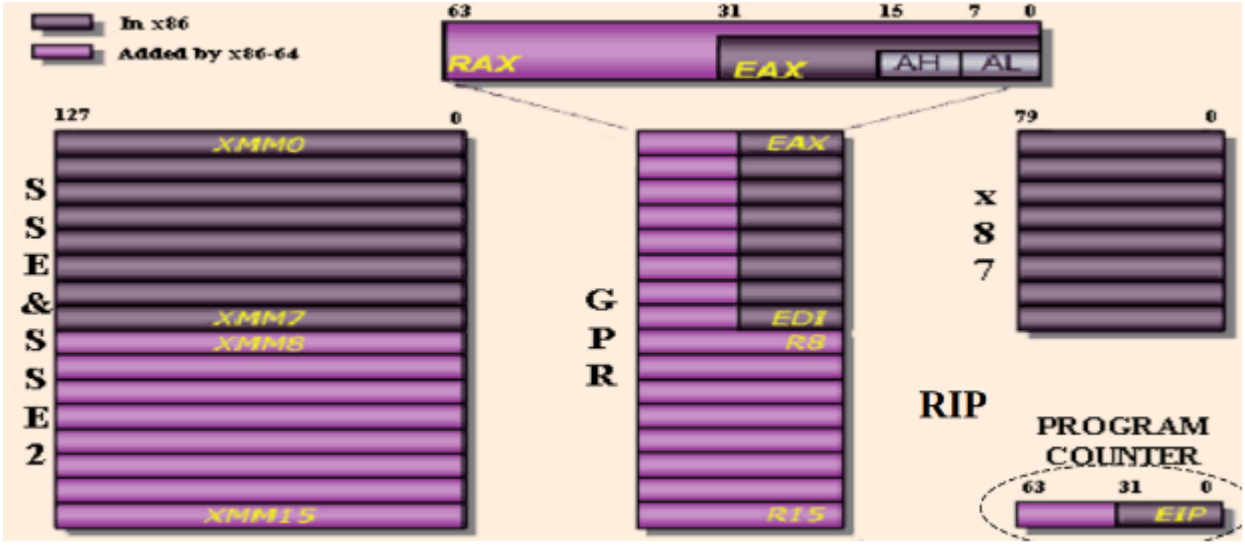
11. Attacco Meltdown

Sappiamo bene che *ciascun processo ha il proprio address space* suddiviso in zone di memoria dedicate all'esecuzione *user mode* e zone di memoria dedicate all'esecuzione *kernel mode*. L'attacco ha come scopo ultimo quello di accedere a un'area di memoria situata nel kernel anche se non si hanno i permessi, magari per andare a leggere delle informazioni sensibili di un utente differente del sistema. Andando nel dettaglio, l'attacco viene eseguito nella maniera spiegata qui di seguito. Anzitutto si definisce un array A nella memoria user space e si svuota la cache tramite l'istruzione `cflush`. Dopodiché, mediante una `mov`, si preleva un **particolare byte** x (*e.g*: address) situato nel kernel e si memorizza il byte in un registro; naturalmente questa `mov` è un'istruzione offending. Si accede alla entry con **spiazzamento** x rispetto all'indirizzo base dell'array A (e.g. caricandone il contenuto in un registro) in modo da farla salire in cache: tale aggiornamento della cache rappresenta proprio il *side effect* lasciato sullo stato micro-architetturale da parte dell'istruzione successiva a quella offending che, chiaramente, viene eseguita *solo speculativamente*. A tal punto, poiché l'array A si trova in user space, è possibile accedere a tutte le sue entry e, per ogni entry, cronometrarne il tempo necessario per l'accesso: la entry relativa al tempo di accesso minore sarà chiaramente quella di spiazzamento x, perché si tratta dell'unica entry il cui contenuto era stato memorizzato in cache. Ed ecco qui: **ora è noto lo spiazzamento x**, che era proprio il byte di memoria prelevato inizialmente dal kernel space. Ricapitolando, il valore x è stato ottenuto in maniera indiretta misurando i tempi di accesso alle informazioni presenti nell'array A. Di conseguenza, abbiamo a che fare con un **side-channel** (o **covert-channel**) **attack**, ovvero con un attacco che fa uso di un canale laterale per andare a rubare delle informazioni. Io parto da un byte *B* presente nella zona kernel, lo salvo in un registro (non potrei), accedo a `array[B]` che va in cache. Successivamente accedo a tutte le entry di array (in un loop continuo, indipendente da *B*). L'accesso più veloce sarà quello ad `array[B]`, e quindi tra tutti gli accessi effettuati, riesco a capire cosa c'è in `array[B]`, perchè più veloce. Quindi prendendo un byte nella zona kernel, riesco da user a leggere qualcosa a livello kernel. Meltdown può essere esteso anche al caso in cui si vuole scoprire un'intera stringa all'interno del kernel space, che può essere una password, una chiave segreta e così via. Negli esempi



12. Insights sui processori x86 e x86-64

X_86 ha 8 registri general purpose, e Program Counter 32 bit.
X_86_64 è backward compatibile, ha 15 registri general purpose a 64 bit e Program Counter a 64 bit.



- **GPR**: registri general purpose.
- **SSE & SSE2**: registri vettoriali, che consentono a singole istruzioni di fare più cose in parallelo.
- **x87**: floating-point stack registers.
- **RIP**: program counter.

Vediamo alcune istruzioni fondamentali per il trasferimento di dati nell'architettura x86-64:

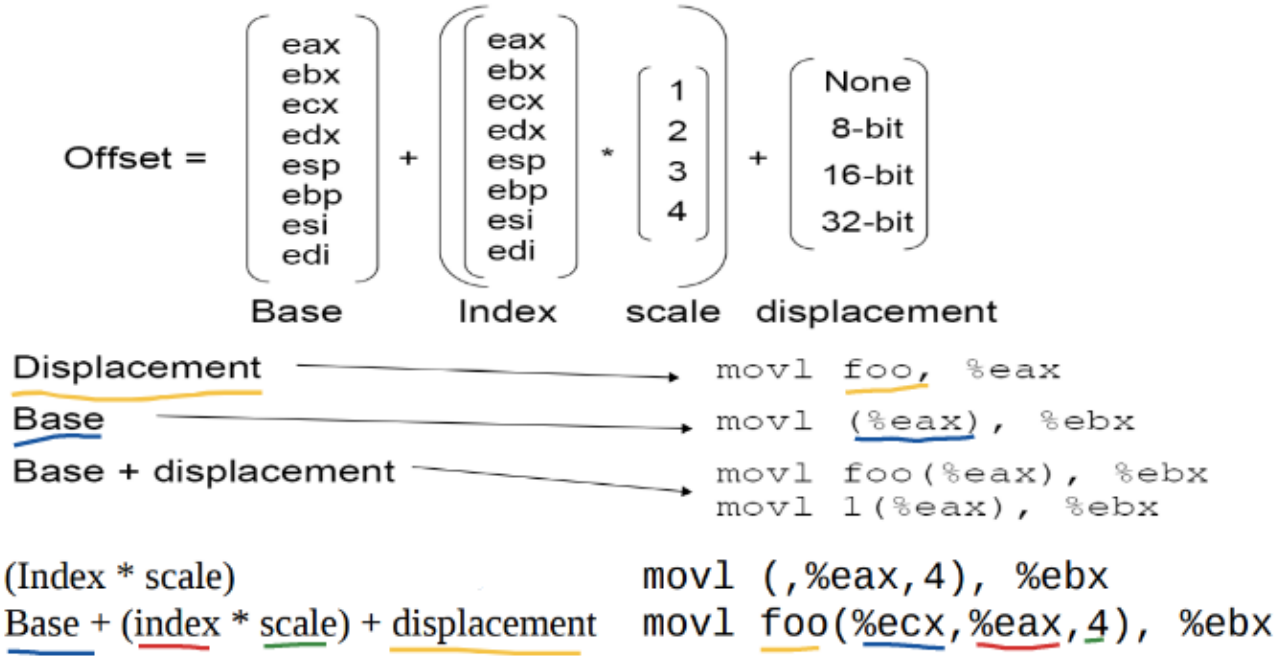
Istruz. (a 64 bit)	Variante a 32 bit	Variante a 16 bit	Variante a 8 bit	Descrizione	Esempio*
mov	movl	movw	movb	Trasferisce un valore su un registro destinazione.	movw $5, %ax
push	pushl	pushw	\	Aggiunge un elemento sullo stack.	pushl %ebx
pop	popl	popw	\	Elimina un elemento dallo stack e lo salva su un registro.	popl %ebx

*Negli esempi specificati in tabella si è adottata la sintassi AT&T che, a differenza di quella Intel, prevede che la sorgente venga posta prima della destinazione.

La parte più complessa però sta nello specificare la sorgente e/o la destinazione **in memoria logica** (e non su un registro). La figura riportata in seguito mostra il meccanismo utilizzato. In *ISA* non ho i nomi delle variabili, solo quelli dei registri, per arrivare ad una variabile di memoria devo usare l'indirizzo di tale variabile in memoria.
Nota: Le componenti *base, index, scale* sono sempre contenute nelle parentesi, ad esempio *movl (..., ..., ...)*

- **Displacement**: può ad esempio essere un indirizzo assoluto noto a tempo di compilazione. Possiamo vederlo come il punto dell'address space che specifica la sorgente (es: *foo è displacement diretto*)
- **Base**: può essere relativo al valore di un pointer.
- **Index*scale**: può rappresentare uno spiazzamento rispetto all'indirizzo base; ad esempio, quando si itera su un array di interi, *scale* vale sempre 4 mentre *index* viene incrementato di 1 a ogni iterazione.

(nb: in %ebc è dove carico il tutto).



Vediamo ora le istruzioni logiche e aritmetiche:

Istruzione (a 64 bit)	Variante a 32 bit	Variante a 16 bit	Variante a 8 bit	Descrizione
and	andl	andw	andb	dest = source && dest
or	orl	orw	orb	dest = source || dest
xor	xorl	xorw	xorb	dest = source ^ dest
not	notl	notw	notb	dest = ^dest
sal	sall	salw	salb	dest = dest << source
sar	sarl	sarw	sarb	dest = dest >> source
add	addl	addw	addb	dest = source + dest
sub	subl	subw	subb	dest = dest – source
inc	incl	incw	incb	dest = dest + 1
dec	decl	decw	decb	dest = dest – 1
neg	negl	negw	negb	dest = ^dest
cmp	cmpl	cmpw	cmpb	Compara due valori; se essi sono uguali, imposta un determinato registro di stato.

13. Codice assembly dell'attacco meltdown

Nella pagina seguente viene mostrato il codice assembly che permette di effettuare l’attacco Meltdown. La sintassi utilizzata è quella di Intel. Chiaramente il codice può essere incapsulato all’interno di un programma C.

```
; rcx = kernel address
; rbx = probe array (array A)
retry:
mov al, byte [rcx] //istr. offending. Prendo byte[rcx] e lo metto in ‘al’, ultimo byte del reg. ‘eax’
shl rax, 0xc      //shifto rax di 0xc = 12, cioè 12 bit a 0 (verso sinistra shifto la parte meno significativa) lo spiazzamento dista 2^12 = 4096 byte dal predecessore SEMPRE.
jz retry          //se il valore letto nel kernel è nullo, cioè rax=0 riprovare.
mov rbx, qword [rbx + rax] //qui si effettua l’accesso al probe array con spiazzamento pari a rax
```