Stalli e performance

Nonappena blocco la pipeline, quindi viene posta in uno **stato di attesa**, le prestazioni vengono ovviamente ridotte, insieme al throughput. Un compilatore creato dalla casa produttrice del processore, riesce ad ottenere dei risultati migliori rispetto agli altri (sullo stesso processore). Questo perchè il compilatore è "a conoscenza" della struttura del processore che sta utilizzando, mentre dei compilatori generici non possono avere.

Previsione dei branch (salti)

I jump dovuti ai Loop sono abbastanza prevedibili, questo perchè nei loop viene ripetuto un percorso numerose volte, mentre si **esce una sola volta**. La tecnica base ha un success rate solo del 60%, quindi inaccettabile; la tecnica che viene invece usata è la **previsione dinamica**, dove si tiene conto delle previsioni precedenti per prendere la decisione (previsione) corrente.

Sui sistemi moderni si tende ad avere una pipeline particolarmente lunga, quindi spezzettando un'istruzione in 10-14 fasi, possiamo elevare la frequenza del processore; quindi se abbiamo una frequenza del clock elevata avremo sicuramente una **pipeline particolarmente lunga**.

Previsione dinamica

La scelta non viene decisa a **tempo di compilazione**, ma viene deciso a tempo **di esecuzione** sulla base di quello che è successo nei salti precedenti; questo significa che dobbiamo tenere traccia delle azioni passate.

Per ogni branch del programma bisogna salvare il suo stato in un buffer apposito, dove è presente una tabella che ci dice per ogni branch le informazioni sui suoi salti precedenti.

Come funziona in realtà?

Se teniamo conto solo dell'ultima "passata" nella realtà la tecnica non funzionerebbe tanto bene;

Supponiamo di avere due loop innestati; il loop più interno viene eseguito 1000 volte e solo una su 1000 il salto non viene preso; se adesso teniamo conto solo dell'ultimo salto, la prossima previsione ci dirà che il salto non dovrebbe essere preso, quando invece il loop dovrà essere eseguito altre 1000 volte.

Eccezioni ed interruzioni

Le interruzioni servono al processore per accorgersi che è avvenuto un evento esternamente, come ad esempio un'operazione di I/O.

C'è una differenza tra **eccezioni ed interruzioni**; le cause che permettono di deviare il processore dal suo loop di esecuzione, possono essere **interne ed esterne**, quindi degli **eventi inaspettati**.

- **Eccezioni** nascono all'interno della CPU: ad esempio ho fornito un'istruzione errata (con un opcode non esistente) e quindi il processore non sa cosa eseguire. Un'altro esempio potrebbe essere una divisione per zero.
  Sono sostanzialmente un richiamo all'attenzione del sistema operativo, o in altre parole, sono delle **system calls (syscalls)** .
  Esistono delle apposite istruzioni che permettono di **attivare il SO** generando un flusso "eccezionale", quindi **sospendo l'esecuzione del programma** e salto altrove (appartenente al SO).
- **Interruzioni** provengono da un controller esterno **I/O** .

Possiamo raggruppare eccezioni ed interruzioni con vari termini, come ad esempio il termine **trap**; questo perchè in entrambi i casi l'esecuzione del processo viene interrotta, ed il controllo passa al sistema operativo; sostanzialmente ciò che accade è il medesimo processo tra le due, cambia solo la causa scatenante.

Le trap sono molto molto frequenti, quindi non è possibile ignorare il loro effetto sul tempo di esecuzione.

Problema con la pipeline

Quando il processore è occupato ad eseguire un processo in pipeline, ed arriva una trap, le istruzioni che erano state completate fino a quel punto, vengono annullate; questo per il processore dovrà eseguire un'altra porzione di codice. Una volta eseguito l'altra porzione di codice, il processore dovrà tornare a dove era rimasto precedentemente.

Come gestire le eccezioni

Solitamente, quando si verifica una trap, bisogna salvare il **program counter** in modo da poter riprendere da quel punto.

Quando abbiamo una trap alcuni processori effettuavano un **jump ad una posizione hardwired**. Nei sistemi moderni, abbiamo la tecnica detta **interrupt vettorizzato**; questo significa che in memoria è presente una tabella che, per ogni **causa di trap**, ci dice dove effettuare il jump.

Interrupt precisi / imprecisi

I sistemi moderni dispongono di **interrupt precisi**, ovvero riescono a mantenere l'istruzione che si stava eseguendo, ed a far si che l'istruzione che viene fermata è **l'istruzione più antica**;

Abbiamo anche dei processori aventi **interrupt imprecisi**, ovvero viene bloccata un'istruzione che non corrisponde esattamente a quella che dovrebbe essere bloccata. Questa tecnica è poco diffusa negli ultimi anni, proprio per le problematiche che si porta dietro.

Parallelismo al livello delle istruzioni (ILP)

Un processore che esegue una sola istruzione per volta è molto lento, abbiamo quindi interesse ad avere un processore che possa eseguire più istruzioni contemporaneamente.

Una delle soluzioni a questo problema è proprio il **pipelining**, come abbiamo visto; le istruzioni non vengono processate sequenzialmente, ma vengono eseguite in parallelo.

Altre tecniche

- **Pipeline più lunghe** : in questo caso abbiamo più istruzioni in esecuzione allo stesso momento, possiamo avere delle frequenze di clock più elevate e throughput più elevato.
  Le pipeline non possono diventare estremamente lunghe: ad esempio un'istruzione semplice com un **add** non può essere frantumata in tantissime fasi diverse.
- **Multiple issue** : realizzo dei processori in cui ci sono delle unità separate che lavorano indipendentemente. Abbiamo quindi più pipelines, le quali lavorano tutte separatamente l'una dall'altra.
  Abbiamo quindi più istruzioni che possono partire contemporaneamente.

Multiple issue

Questa tecnica viene usata abbastanza, anche se i risultati non sono molto ingenti. L'idea è quella di mandare in esecuzione più istruzioni alla volta, e lo si può fare in due modi:

`Piccolo disclaimer:` Tutte le cose statiche sono molto più ragionevoli da implementare, questo perchè le cose dinamiche vengono fatte a tempo di esecuzione, quindi il processore deve eseguire dei task **in hardware**.

Statico

Con questa tecnica, il compilatore "decide" di organizzare delle istruzioni **a gruppi** in modo che queste possano partire contemporaneamente; quindi capiamo che è il **compilatore** che tenta di capire dove sono i possibili **hazards** e non mettere insieme delle istruzioni che potrebbero generarne uno, e quindi le posizionerà in "gruppi diversi" (come quando alle elementari facevi comunella con il tuo amichetto e le maestre vi dividevano).

Dinamico

Il compilatore, però, non può far fronte a tutti i problemi eventuali a tempo dell'esecuzione, proprio perchè il compilatore non è a conoscenza dei dati su cui il processo lavorerà a tempo di esecuzione.

Via hardware è molto più complicato implementare il tutto, ma è molto più efficiente; questo perchè a tempo di esecuzione si è a conoscenza dei dati su cui si sta lavorando, ed esaminando la situazione possiamo capire cosa può avvenire contemporaneamente e cosa no.

In questo caso è la **CPU a risolvere gli hazards** con delle tecniche complicate, a tempo di esecuzione; in questo caso il compilatore (software) non mette più mano.

Speculazione

In qualche modo si tende ad "indovinare" se un'istruzione deve essere eseguita o meno. Se la speculazione era giusta, salvo il risultato dell'istruzione, se invece l'istruzione non doveva essere eseguita, abbiamo un **rollback** ed eseguo l'istruzione giusta.

Per "istruzione" si intende, ad esempio, un jump a seguito di un branch, come abbiamo visto nella lezione precedente.

Questa tecnica viene usata sia nel caso statico che dinamico ; un'altra operazione speculativa che si potrebbe fare è sui caricamenti della memoria:

Facciamo l'ipotesi che dalla memoria il dato non è stato modificato rispetto all'ultima volta; se invece si scopre che il valore era stato modificato, si esegue il **rollback**.

Static multiple issue - esteso

Possiamo vedere i vari pacchetti come un'unica grande istruzione, infatti il nome storico per questo tipo di processori, è di **processori VLIW**, ovvero very long instruction word.

Abbiamo quindi delle word di istruzioni particolarmente lunghe, proprio perchè sono tante piccole istruzioni impacchettate in un'unica grande istruzione.

Alla domanda: cosa è un processore VLIW? E' un processore con **multiple issue statico**, in cui il compilatore genera mega istruzioni in cui sono impacchettate diverse istruzioni che possano partire contemporaneamente.

Il **compilatore** deve rimuovere gli hazards, riordinare le istruzioni e comporre i pacchetti in modo che non ci siano dipendenze all'interno del pacchetto (le istruzioni non devono darsi fastidio tra di loro).
Quando non è possibile **trovare delle istruzioni da far partire contemporaneamente**, il compilatore si arrende ed inserisce delle **nop**, ovvero **no-operation**. Questa istruzione non fa assolutamente niente.

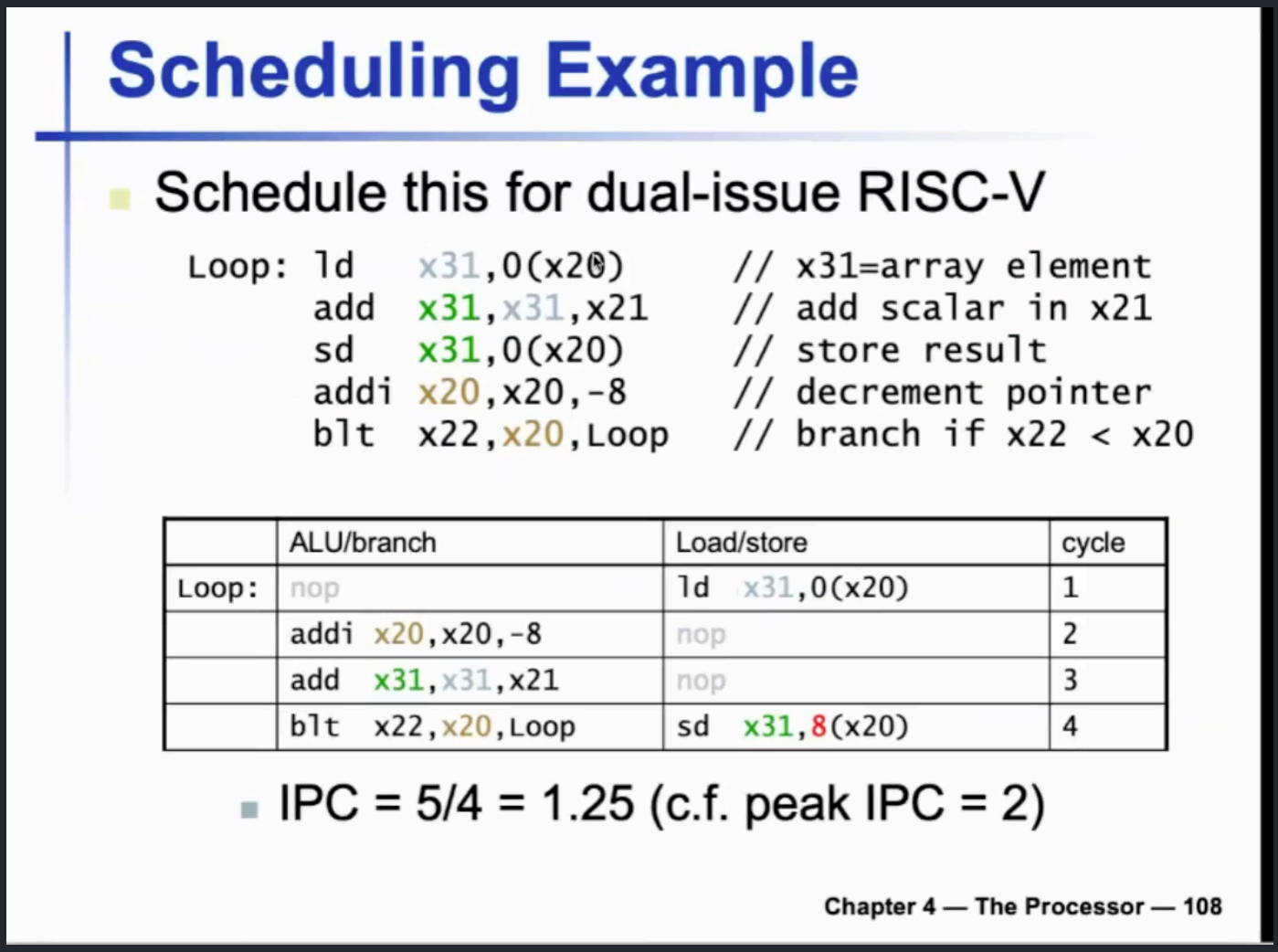
Hazards con il dual issue statico in RISC-V

Cosa succede quando componiamo i pacchetti delle istruzioni?
Fondamentalmente non possiamo fare. un add ed un load contemporaneamente:

```
add x10, x0, x1
ld x2, 0(x10)
```

Quest istruzioni prende il risultato di `x0+x1` e lo salva in x10. L'istruzione successiva invece carica un valore da `0(x10)` (memoria) e lo scrive in x2.
Inevitabilmente, queste due istruzioni **devono essere poste in due pacchetti separati**.

Esempio



In questo loop abbiamo un `ld, add, sd, ...`; stiamo sommando tutti gli elementi in un array.

Riusciamo sempre a creare un pacchetto di due istruzioni nel quale uno è ALU/branch e l'altro è Load/store? Come possiamo vedere nella foto, non riusciamo.

Prima di caricare x31, finchè esso non è stato caricato, non possiamo effettuare la somma che usa x31 come operando. Come possiamo vedere, questa operazione è posta al **primo ciclo di clock** del loop. Siccome questa operazione non viene eseguita in un unico ciclo di clock (ha bisogno di diversi cicli di clock per essere completata), quello che si può fare è **decrementare x20** (che non è intaccato da x31):

`addi x20, x20, -8`; come possiamo vedere dall'immagine, questa istruzione non era stata originariamente posta subito dopo il ld, ma alcune istruzioni dopo.

Finalmente, al terzo ciclo di clock possiamo utilizzare il registro x31, e quindi effettuiamo la somma; ma non possiamo abbinarla a nessuna istruzione **load/store** perchè finchè l'operazione `x31+x21` non è stata completata, in x31 è presente ancora il valore precedente, e quindi non può essere salvato.

Alla fine, all'ultimo colpo di clock (4) possiamo effettuare sia un'operazione CPU che una di Store, andando ad effettuare un **branch ed il salvataggio di x31 in memoria**.

Morale della favola

Su 4 pacchetti, solo una volta il compilatore è riuscito a mettere in un unico pacchetto entrambe le operazioni di CPU e Store.

Loop Unrolling

Un compilatore può tentare di usare una tecnica che sfrutta meglio il parallelismo.

Supponiamo di avere un classico programma con un ciclo for:

```
for(i=0; i<100; i++){
  ...
}
```

in assembly (fantastico) possiamo scriverlo nel tipo:

```
  i = 0
loop:
  ...
  branch jump loop
```

Ragionando, la parte utile del codice è solo la parte contenuta **all'interno** del for, poi tutta la parte di instanziamento, branch, ecc. sono solo operazioni di tipo **overhead**, ovvero tempo "buttato".

Il tempo speso per il controllo è pesato rispetto al lavoro utile svolto; questo vuol dire che se ad esempio faccio mezz'ora di auto per andare al supermercato per prendere una bottiglietta d'acqua, non ne vale la pena; se però invece la spesa è molto ingente, il gioco vale la candela.

Lo stesso ragionamento vale per i loop: se all'interno del loop ci sono molte operazioni, l'overhead è sopportabile; se invece all'interno è posta solo un'operazione, magari anche stupida, allora non ne vale più la pena.

Esempio: tanto overhead

```
for(i=0; i<100; i++)
  x[i] = 0;
```

Esempio: overhead ridotto

```
for(i = 0; i<100; i+=4){
  i[i] = 0;
  i[i+1] = 0;
  i[i+2] = 0;
  i[i+3] = 0;
}
```

Questo loop non verrà più percorso 100 volte, ma solo 25, ma il risultato è lo stesso; questo significa **srotolare il loop**.

---

fine lezione 14