

# Pipeline in una macchina reale

Ing. Fabio Fassetti

DEIS - Università della Calabria

*ffassetti@deis.unical.it*

## Introduzione

La pipeline è una tecnica in cui parti appartenenti a istruzioni distinte vengono eseguite contemporaneamente. Questa tecnologia viene utilizzata dai microprocessori per incrementare il *throughput*, ovvero la quantità di istruzioni eseguite in una data quantità di tempo. L'elaborazione di un'istruzione da parte di un processore si compone di operazioni elementari. Ogni CPU è gestita da un clock centrale e ogni operazione elementare richiede almeno un ciclo di clock per poter essere eseguita. Le prime CPU erano formate da un'unità polifunzionale che svolgeva cinque passaggi legati all'elaborazione delle istruzioni:

1. **IF**: Lettura dell'istruzione da memoria
2. **ID**: Decodifica istruzione e lettura operandi da registri
3. **EX**: Esecuzione dell'istruzione
4. **MEM**: Attivazione della memoria (solo per certe istruzioni)
5. **WB**: Scrittura del risultato nel registro opportuno

Una CPU classica quindi richiedeva almeno cinque cicli di clock per eseguire una singola istruzione [Figura 1(a)].

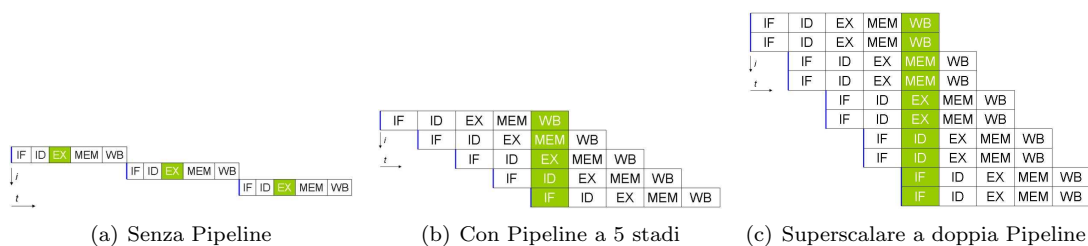


Figura 1: Flusso temporale delle istruzioni

Con il progresso della tecnologia si è potuto integrare un numero maggiore di transistor in un microprocessore e quindi si sono potute parallelizzare alcune operazioni riducendo i tempi di esecuzione. La pipeline dati è la massima parallelizzazione del lavoro di un microprocessore. In Figura 1(b) vi è un esempio di CPU con Pipeline a 5 stadi. La CPU lavora come in una catena di montaggio ed è quindi composta da 5 unità, ognuna delle quali provvede a svolgere solo un compito specifico che rappresenta uno stadio della Pipeline. Lavorando come una catena di montaggio quando la catena è a regime ad ogni ciclo di clock esce un'istruzione completata. Nello stesso istante ogni unità sta processando una diversa istruzione in parallelo. In sostanza si guadagna una maggior velocità di esecuzione pagando una maggior complessità circuitale del microprocessore che non deve essere più composto da una sola unità ma da cinque unità che devono collaborare tra loro. Per realizzare CPU con prestazioni migliori col tempo si è affermata la strategia di integrare

in un unico microprocessore più pipeline che funzionano in parallelo. Questi microprocessori sono definiti superscalari dato che sono in grado di eseguire mediamente più di un'operazione per ciclo di clock [Figura 1(c)]. Il semplice fatto di eseguire più istruzioni nello stesso ciclo di clock non rende una CPU superscalare: una CPU con una pipeline semplice, che può quindi caricare un'istruzione, eseguirne un'altra e immagazzinare il risultato di quella ancora precedente non è necessariamente superscalare.

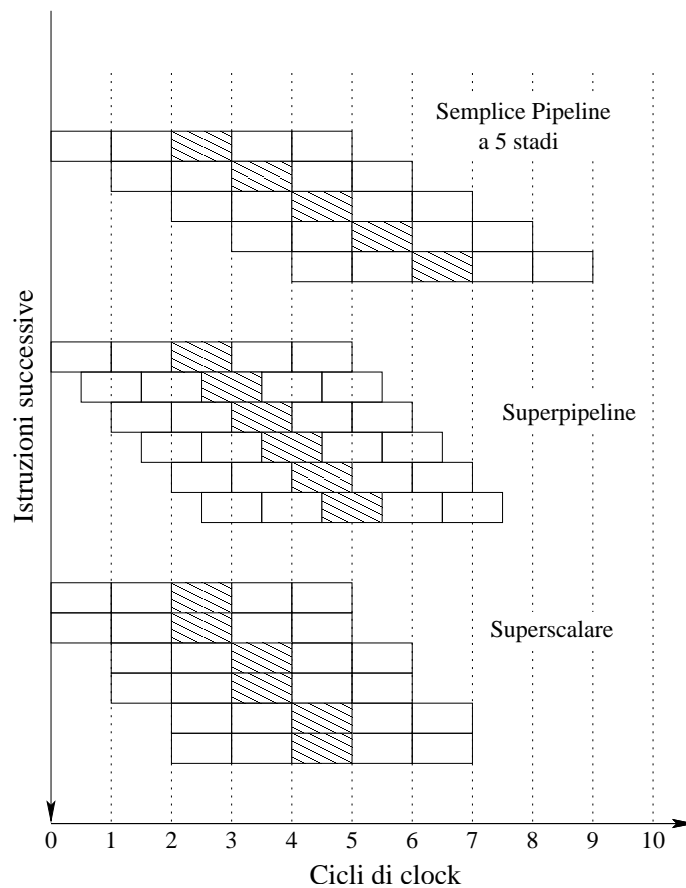


Figura 2: Confronto tra architetture

In una CPU superscalare sono presenti diverse unità funzionali dello stesso tipo, con dispositivi addizionali per distribuire le istruzioni alle varie unità. Per esempio, sono generalmente presenti numerose unità per il calcolo intero (ALU). Le unità di controllo stabiliscono quali istruzioni possono essere eseguite in parallelo e le inviano alle rispettive unità.

L'implementazione di una pipeline non sempre moltiplica il throughput. Questo è dovuto al fatto che le istruzioni possono richiedere l'elaborazione di dati non ancora disponibili e alla presenza dei salti condizionati.

- Il primo problema deriva dal lavoro parallelo delle unità. Supponiamo che la CPU con pipeline debba eseguire il seguente frammento di codice:

$$C := A + B$$

$$D := C - 1$$

La prima istruzione deve prelevare i numeri contenuti nelle variabili A e B, sommarli e porli nella variabile C. La seconda istruzione deve prelevare il valore contenuto nella variabile C, sottrarlo di uno e salvare il risultato in D. Ma la seconda istruzione non potrà essere elaborata (EX) fino a quando il dato della prima operazione non sarà disponibile in memoria (MEM) e quindi la seconda operazione dovrà bloccarsi per attendere il completamento della prima e quindi questo ridurrà il throughput complessivo.

- Il secondo problema è relativo ai salti condizionati. I programmi contengono delle istruzioni condizionate, ossia se una specifica condizione è verificata provvedono a interrompere il flusso abituale del programma e a mandare in esecuzione un altro pezzo di programma indicato dall'istruzione di salto. Ogni volta che questo accade il microprocessore si trova a dover eseguire un nuovo flusso di operazioni e quindi deve svuotare la pipeline del precedente flusso e caricare il nuovo flusso. Ovviamente queste operazioni fanno sprecare cicli di clock e quindi deprimono il throughput. Per ridurre questo problema le CPU adottano delle unità chiamate unità di predizione delle diramazioni (in inglese Branch Prediction Unit) che fanno delle previsioni sul flusso del programma. Queste unità riducono notevolmente i cicli persi per i salti.

Nelle CPU moderne inoltre le pipeline non sono composte da soli cinque stadi ma ne utilizzano molti di più (il Pentium 4 ne utilizza 20). Queste pipeline ovviamente rendono ancora più complesso la gestione dei problemi di coerenza e dei salti condizionati. Pipeline così lunghe si sono rese necessarie per potere innalzare la frequenza di clock. Spezzettando le singole operazioni necessarie per completare un'istruzione in tante sotto operazioni si può elevare la frequenza della CPU dato che ogni unità deve svolgere un'operazione più semplice e quindi può impiegare meno tempo per completare la sua operazione. Questa scelta di progettazione consente effettivamente di aumentare la frequenza di funzionamento delle CPU ma rende critico il problema dei salti condizionati. In caso di un salto condizionato non previsto il Pentium 4 per esempio può essere costretto a svuotare e ricaricare una pipeline di 20 stadi perdendo fino a 20 cicli di clock contro una classica CPU a pipeline a 5 stadi che avrebbe sprecato nel peggiore delle ipotesi 5 cicli di clock.

## 1 Panoramica sulla microarchitettura NetBurst

NetBurst è il nome della micro-architettura erede della P6 nella famiglia di CPU x86 di Intel su cui si basa il Pentium 4.

Esternamente il Pentium 4 appare come una tradizionale macchina CISC, con un vasto set di istruzioni, che supportano operazioni intere a 8-, 16- e 32-bit, e operazioni floating-point a 32- e 64-bit. In realtà, però, la sua architettura ha un potente cuore RISC: ogni istruzione viene, infatti, suddivisa in una sequenza di una o più micro-operazioni RISC, eseguite, attraverso una lunga pipeline (almeno 20 stadi), a velocità di clock molto elevate. Le più importanti caratteristiche introdotte sono la "Hyper Pipelined Technology", il "Rapid Execution Engine" e la "Execution Trace Cache".

- **Hyper Pipelined Technology:** questo è il nome che Intel ha scelto per la pipeline a venti stadi prevista dall'architettura NetBurst. Questo è un aumento significativo, paragonato ai soli 10 stadi della pipeline del Pentium III. Una pipeline così lunga ha comunque degli svantaggi, in particolare il gran numero di stadi che devono essere ripercorsi nel caso in cui l'algoritmo di *branch prediction* faccia un errore (cache miss). Per limitare i danni (miss penalty) dovuti a tali inevitabili problemi, Intel ha introdotto le tecnologie Rapid Execution Engine e Execution Trace Cache e ha raffinato l'algoritmo di branching, migliorando notevolmente la hit rate.
- **Rapid Execution Engine:** Intel ha aggiunto due unità per le operazioni con gli interi rispetto all'architettura P6. Le aggiunte sono un addizionale per interi e un'unità di calcolo per gli indirizzi. Ma la novità più importante introdotta da questa tecnologia è la velocità di clock della ALU, che opera al doppio della velocità di clock del core. Questo vuol dire che in una CPU a 3 GHz la ALU opera a 6 GHz. Queste caratteristiche migliorano di molto le prestazioni della CPU nei calcoli sugli interi.
- **Execution Trace Cache:** Intel ha incorporato una Execution Trace Cache. Questa cache memorizza le micro-operazioni dopo lo stadio di decode, cosicché quando deve passare a una nuova operazione, invece di dover eseguire di nuovo il fetching e il decoding dell'istruzione, la CPU può accedere direttamente alle micro-operazioni dalla trace cache, risparmiando una notevole quantità di tempo. Inoltre le micro-operazioni sono mantenute nella cache secondo

l'ordine di esecuzione predetto algoritmicamente, il che significa che quando la CPU recupera le istruzioni dalla cache, esse sono già presenti nell'ordine corretto.

Le operazioni del Pentium 4 possono essere riassunte come segue:

1. Il processore carica le istruzioni dalla memoria nell'ordine dettato dal programma (codice scritto dal programmatore o da un compilatore).
2. Ogni istruzione viene tradotta in una sequenza di istruzioni RISC, dette *micro-operazioni* o *micro-ops*.
3. Il processore esegue le micro-ops utilizzando un'organizzazione a pipeline superscalare, in cui le micro-ops possono essere eseguite disordinatamente.
4. Il processore esegue il *commit* dei risultati di ogni micro-op sul set dei registri architetturali nello stesso ordine del flusso del programma originale.

Il Pentium 4 consiste di quattro principali sottosezioni: “memory subsystem”, “front end”, “out-of-order control”, “execution units”.

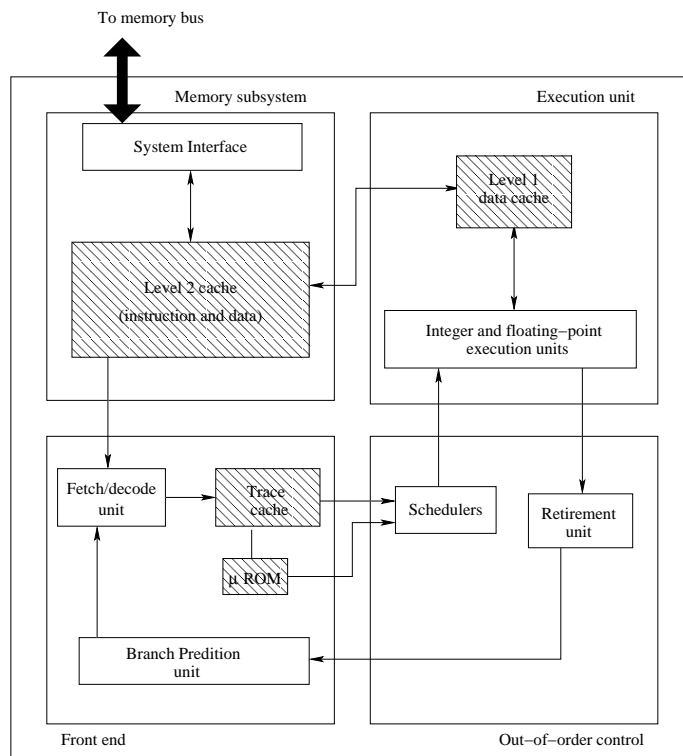


Figura 3: Diagramma a blocchi del Pentium 4

Il “memory subsystem” contiene la logica per l’accesso alla RAM esterna tramite il *memory bus* e una cache di livello 2; nella prima generazione di Pentium 4 era di 256KB, nella seconda di 512KB, nella terza di 1MB. Associata alla cache vi è un’unità di prefetch (non mostrata in figura) che carica i dati dalla memoria principale in L2 prima che essi servano. Da L2 i dati possono essere trasferiti ad altre cache ad elevate velocità.

Sotto il “memory subsystem” in Figura 3, c’è il “front end” che carica le istruzioni secondo l’ordine del programma da L2 e le decodifica. Ogni istruzione viene suddivisa in una sequenza di micro-ops RISC. Per le operazioni semplici (da 1 a 4 micro-istruzioni) l’unità “fetch/decode” determina la sequenza necessaria; per quelle più complesse, la sequenza viene cercata nella micro-ROM. Quindi in entrambi i casi ogni istruzione del Pentium 4 viene convertita in una sequenza di micro-ops che verranno eseguite dal cuore RISC del chip.

Le micro-istruzioni decodificate vengono caricate nella "trace cache", che è la cache di livello 1 per le istruzioni. L'approccio di memorizzare in cache delle istruzioni già decodificate è una delle differenze chiave tra l'architettura NetBurst e la precedente P6. La *branch prediction* viene effettuata qui.

Le istruzioni vengono estratte dalla "trace cache" e inviate agli "scheduler". Quando si incontra un'istruzione che non può essere eseguita, lo scheduler la trattiene e continua a processare il flusso di istruzioni. Vengono eseguite operazioni di rinominazione dei registri per permettere l'esecuzione di istruzioni con dipendenza WAR o WAW.

Sebbene le istruzioni possano essere eseguite in disordine, è necessario che il *commit* dei risultati di ogni micro-ops venga effettuato nell'ordine dettato dal programma e questo è assicurato dalla "retirement unit". L'importanza di ciò è sottile ma rilevante. Supponiamo che le istruzioni vengano completate in disordine. Se occorresse un'interrupt, sarebbe molto difficile salvare lo stato per poter riprendere l'esecuzione successivamente; in particolare sarebbe impossibile dire che le istruzioni precedenti a un dato indirizzo sono state eseguite e le successive no.

Infine nel quarto quadrante ci sono le unità esecutive, che lavorano in parallelo e prendono i loro dati dai registri e da una cache dati di livello 1.

## 2 Pipeline nell'architettura NetBurst

La pipeline del Pentium 4 è la seguente:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC next IP		TC fetch		Drv	All	Rename		Que		Scheduling		Disp		RF		Ex	Flgs	BC	Drv

TC Next IP = trace cache next instruction pointer  
 TC Fetch = trace cache fetch  
 Drv = drive  
 All = allocate  
 Rename = register renaming  
 Que = micro-op queuing  
 Scheduling = micro-op scheduling  
 Disp = Dispatch  
 RF = register file  
 Ex = execute  
 Flgs = flags  
 BC = branch check

Figura 4: Pipeline del Pentium 4

La descrizione dei 20 stadi verrà fatta illustrata utilizzando lo schema di figura 5, del tutto simile a quello di figura 3 ma con più dettagli.

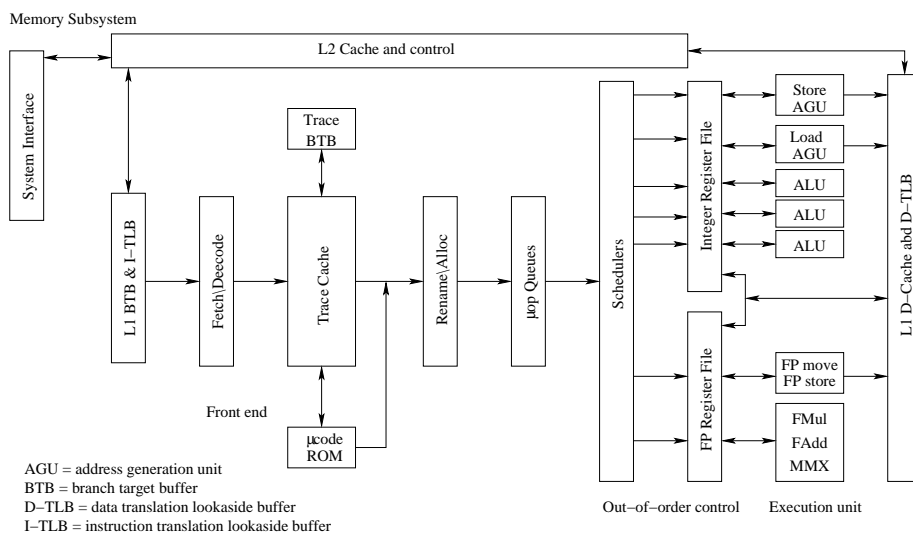


Figura 5: Diagramma a blocchi del Pentium 4

## Front End

**Generation of Micro-Ops** Il front end è la parte della macchina che si occupa di prelevare le prossime istruzioni da eseguire e prepararle per la Pipeline. Il suo lavoro è fornire un ampio flusso di istruzioni al cuore “out-of-order” del processore. Si occupa di caricare le istruzioni dalla memoria in una cache L1, chiamata *Trace Cache*, da cui propriamente comincia la Pipeline, e prepararle per l'esecuzione. Il processore utilizza normalmente la trace cache, quando viene richiesta un'istruzione non presente (cache miss), il front end si occupa di caricare nuove istruzioni nella TC.

L'unità fetch/decode scorre i byte per determinare i limiti delle istruzioni. Quest'operazione è necessaria a causa della lunghezza variabile delle istruzioni del Pentium 4. Il decoder traduce ogni istruzione-macchina in micro-ops (da 1 a 4), ognuna delle quali è un'istruzione RISC a 118-bit. Micro-ops così lunghe sono necessarie per realizzare le complesse operazioni del Pentium, ma sono, in ogni caso, molto più facili da gestire delle istruzioni originali da cui derivano. Le micro-ops così generate vengono memorizzate nella TC. Se invece un'istruzione richiede più di 4 micro-ops non viene decodificata nella trace cache, ma viene segnalato all'unità di controllo di cercare nella micro-ROM.

Se il decoder incontra un'istruzione di salto condizionato, viene usata una strategia di predizione dinamica, basata sulla storia delle istruzioni di salto eseguite recentemente. Si cerca nell'L1 BTB (L1 - Branch Target Buffer) se l'istruzione sia stata già predetta di recente, e in caso positivo, si continua dall'indirizzo predetto. L'L1 BTB serve quindi a memorizzare le informazioni riguardanti le istruzioni di salto incontrate di recente, ed ha una capacità di circa 4KB. Se un'istruzione di salto condizionato non è presente viene aggiunta al BTB eventualmente cancellando la meno recente, e la predizione viene effettuata utilizzando un algoritmo statico, in accordo alle seguenti regole:

1. un salto “all'indietro” viene assunto come parte di un ciclo e quindi viene considerato, cioè il prossimo indirizzo sarà quello dettato dal salto. L'accuratezza di questa predizione statica è molto elevata.
2. un salto “in avanti”, invece, viene considerato come uscita precoce da un ciclo, o come condizione di un “if-statement” e non viene accettato; cioè la prossima istruzione ad essere seguita sarà quella immediatamente successiva all'istruzione di salto. L'accuratezza di questa è molto più bassa della precedente.

**Trace Cache Next Instruction Pointer** I primi due stadi della Pipeline si occupano della selezione delle istruzioni nella trace cache e utilizzano un meccanismo di predizione delle diramazioni separato da quello descritto precedentemente. Viene a tale scopo mantenuto un buffer (*Trace BTB*, trace - branch target buffer) più piccolo dell'L1 BTB, in quanto il suo scopo è di predire i salti solo all'interno delle istruzioni del programma contenute in TC. Ogni qual volta si incontra un'istruzione di salto nel flusso di istruzioni viene controllato il BTB, se è presente, il salto viene accettato o meno in funzione della storia memorizzata nel BTB.

**Trace Cache Fetch** La trace cache memorizza le micro-ops già decodificate dal decoder, le istruzioni che richiedono più di 4 micro-ops vengono trasferite dalla *microcode ROM*, che contiene la sequenza di micro-ops (5 o più) associate con istruzioni macchina complesse. Dopo che la ROM ha terminato la sequenza di micro-ops necessarie all'istruzione corrente, la prossima operazione di fetch dell'istruzione riparte dalla TC.

**Drive** Il quinto stadio del Pipeline trasferisce le micro-ops dalla TC al modulo rename/allocator (tre micro-ops per ogni colpo di clock).

## Out-of-order Control

Questa parte del processore riordina le micro-ops per permettere che vengano eseguite il prima possibile.

**Allocate** Questo stadio alloca le risorse necessarie per l'esecuzione delle istruzioni. Per tale scopo compie le seguenti funzioni:

- Crea, per ogni micro-ops, una voce nel ROB (*ReOrder Buffer*) che tiene traccia del loro stato fino al completamento.
- Controlla se le risorse necessarie alla micro-op, ad esempio registri, sono disponibili.
- In caso positivo accoda la micro-op a una delle code che precedono gli scheduler. Vengono mantenute code separate per micro-ops in memoria e non in memoria.
- Se una micro-op non può essere eseguita viene differita e vengono elaborate le successive, conducendo a un'esecuzione delle micro-ops in disordine. Questa strategia è necessaria per far sì che le unità esecutive siano sempre, per quanto possibile, impegnate, così da ottimizzare le performance del chip.

Il ROB è un buffer circolare che può memorizzare fino a 126 voci e contiene anche 128 registri hardware. Ogni entry del ROB è relativa a una micro-op e memorizza i seguenti campi ad essa relativi:

- **State:** Lo stato della micro-op, cioè se è in coda per l'esecuzione, è in esecuzione, o la sua esecuzione è terminata ed è in attesa del *commit*.
- **Memory Address:** L'indirizzo dell'istruzione Pentium originale da cui deriva la micro-op.
- **Micro-op:** La micro-op relativa a questa entry.
- **Alias Register:** Se la micro-op fa riferimento ad uno dei 16 registri architetturali (EAX, EBX, ECX, ecc.), viene scritto qui il reindirizzamento di questo riferimento ad uno dei 128 registri hardware effettuato al passo successivo.

Le micro-op entrano nel ROB in ordine, poi vengono inviate all'unità Dispatch/Execute in disordine. Il criterio è quello di inviare alle unità esecutive le micro-ops che hanno disponibili l'unità esecutiva appropriata e tutti i dati.

**Register Renaming** Lo stadio di ridenominazione mappa i riferimenti ai registri architetturali (8 floating-point più EAX, EBX, ECX, EDX, ESI, EDI, EBP e ESP) nel set dei 128 registri hardware. Questo stadio quindi rimuove false dipendenze causate dal limitato numero di registri architetturali (quindi spesso elimina dipendenze WAR e WAW), preservando però le dipendenze reali (dipendenze RAW).

#### Esempio.

Consideriamo la seguente porzione di codice:

$$\begin{aligned} I_1 : R_7 &= R_1 * R_2 \\ I_2 : R_1 &= R_0 - R_2 \\ I_3 : R_3 &= R_3 * R_1 \\ I_4 : R_1 &= R_4 + R_4 \end{aligned}$$

$I_2$  calcola il valore di  $R_1$  poi utilizzato da  $I_3$ . Questo valore non sarà più utilizzato perché  $I_4$  lo sovrascrive. La scelta di utilizzare  $R_1$  come registro intermedio, sebbene perfettamente plausibile per un programmatore, che immagina sempre che le istruzioni vengano eseguite in sequenza, è una pessima scelta. Infatti,  $I_2$  e  $I_4$  non possono essere eseguite prima di  $I_1$  perché altrimenti il valore di  $R_1$  letto sarebbe scorretto (WAR), quindi neanche  $I_3$ , perché ha bisogno del valore di  $R_1$  modificato da  $I_2$  (RAW). Però, nessuna di queste tre istruzioni ha realmente bisogno che venga eseguita  $I_1$  prima, in quanto nessuna utilizza  $R_7$ . Sfruttando la tecnica di "register renaming" si può risolvere facilmente questo problema, il codice diventa:

$$\begin{aligned} I_1 : R_7 &= R_1 * R_2 \\ I_2 : S_1 &= R_0 - R_2 \\ I_3 : R_3 &= R_3 * S_1 \\ I_4 : S_2 &= R_4 + R_4 \end{aligned}$$

dove  $S_1$  e  $S_2$  sono registri segreti (non visibili al programmatore).

In questa nuova scrittura il risultato finale è esattamente identico a quello che si otterrebbe con il codice di sopra, però ora l'unica dipendenza rimasta è la dipendenza tra  $I_2$  e  $I_3$ , cioè l'unica reale, in cui  $I_3$  deve aspettare  $I_2$  (dipendenza RAW). ■

**Micro-Op Queuing** Dopo l’allocazione delle risorse e la ridenominazione dei registri, le micro-ops vengono accodate in attesa di essere schedulate. Ci sono due code per le operazioni in memoria (load e store) e una per quelle che non coinvolgono la memoria. Ogni coda gestisce le micro-ops strettamente in ordine FIFO, rispetto a quelle presenti sulla propria coda, ma non rispetto alle altre code. Quindi tra operazioni presenti su code diverse, non vi è alcun ordine stabilito.

**Micro-Op Scheduling e Dispatching** Gli scheduler sono responsabili di prelevare le micro-ops dalle code e inviarle per l’esecuzione, è loro compito controllare le micro-ops per verificare quali possano essere eseguite. Questo è il cuore del motore “out-of-order”. Gli scheduler fanno sì che le istruzioni vengano eseguite appena possibile. Nel caso più operazioni fossero pronte per l’esecuzione, verrebbe rispettato l’ordine FIFO delle code. Questa è una sorta di disciplina FIFO in favore di un’esecuzione “in ordine”.

Gli scheduler sono collegati a 4 diversi tipi di porte per l’invio delle micro-ops alle unità operative. Una porta per l’invio di micro-ops di “memory load”, una per quelle di “memory store”, e due per il calcolo. Tramite queste ultime due possono essere inviate fino a 2 operazioni per ciclo, tramite le prime, invece, solo una per ciclo. Quindi in totale l’insieme delle porte può inviare fino a 6 micro-ops per colpo di clock alle unità operative.

## Execution Units

**Integer e Floating Point Execution Units** Le unità esecutive sono progettate per ottimizzare il rendimento generale della macchina, quindi rendendo le operazioni più comuni il più veloce possibile. Vi sono due ALU a bassa latenza per operazioni intere semplici (add/sub/logic), che operano a due volte la frequenza di clock, un’ALU per operazioni intere complesse (mul/div), 2 unità per le operazioni floating-point e MMX, e, infine, 2 unità rispettivamente per le operazioni di load e store in memoria. Gli “integer/floating-point register” rappresentano le sorgenti degli operandi di cui necessitano le micro-ops per essere eseguite. Queste ultime possono leggere i dati anche dall’*L1 Data Cache*.

Uno stadio separato della Pipeline è dedicato al calcolo dei **flags** (zero, negative, ...), che sono solitamente gli input di istruzioni di salto.

Il successivo stadio è quello di controllo delle diramazioni (**Branch Checking**). In questo stadio viene effettuato il controllo tra l’attuale risultato della diramazione con la predizione. In caso di predizione scorretta, le micro-ops devono essere rimosse da vari stadi della Pipeline.

Nel successivo e ultimo stadio di **drive** la diramazione effettiva viene inviata al BTB che ripristinerà la Pipeline a partire dal nuovo indirizzo corretto.

## Riferimenti bibliografici

W. Stallings, *Computer Organization & Architecture - Designing for Performance - 7<sup>a</sup> edizione*, 2006

A. S. Tanenbaum, *Structured Computer Organization - 5<sup>a</sup> edizione*, 2006

G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, P. Roussel, *The Microarchitecture of the Pentium 4 Processor*, Intel Technology Journal Q1, 2001