La pipeline

G. Lettieri

13 Maggio 2025

In quest'ultima parte del corso ritorniamo a concentrarci sulla CPU. In particolare, esaminiamo alcuni aspetti avanzati dell'architettura che permettono di eseguire il flusso di istruzioni più velocemente, sfruttando le opportunità di svolgere più azioni contemporaneamente. Faremo sempre riferimento ai processori Intel, limitatamente a ciò che avviene all'interno di un singolo "core" (quindi, un unico flusso di istruzioni). Rispetto al resto del corso rimarremo però ad un livello più astratto e meno aderente ai dettagli della CPU reale, che in gran parte non sono noti (anche se molti sono stati dedotti a-posteriori tramite *reverse engineering*).

La prima e più antica tecnica che permette di velocizzare l'esecuzione di un flusso di istruzioni è quella della *pipeline* (catena di montaggio). In analogia con le catene di montaggio industriali, l'idea è di pensare alla CPU come una fabbrica che produce i risultati delle istruzioni. Al solito, conviene concentrarsi inizialmente sulle operazioni aritmetico/logiche, che hanno due operandi e producono un risultato. Per produrre questi risultati, ogni istruzione deve passare attraverso varie fasi di "lavorazione": l'istruzione deve essere prelevata dalla memoria, decodificata, devono essere prelevati i suoi operandi, deve essere eseguita, e infine il risultato deve essere scritto nella destinazione.

Per esempio, pensiamo all'istruzione assembly

```
add %rax, 16(%rbx, %rcx, 8)
```

In memoria, l'istruzione è codificata con i byte `48 01 44 cb 10`. Questi byte devono essere prelevati dalla memoria e portati in un registro interno del processore; quindi devono essere decodificati per capire che si tratta di una operazione di somma con due operandi: il primo si trova nel registro `rax` e il secondo si trova in memoria, ad un indirizzo che deve essere calcolato sommando 16 al contenuto di `rbx` e al contenuto di `rcx` moltiplicato per 8; una volta prelevati i due operandi, deve essere eseguita la somma e il risutato deve essere infine scritto in memoria allo stesso indirizzo calcolato in precedenza. Ora, possiamo immaginare che la rete logica che esegue la somma (la ALU) sia completamente distinta dalla rete logica che decodifica l'istruzione; quindi, per tutto il tempo in cui la CPU sta calcolando la somma, la rete di decodifica resta inutilizzata. L'idea è di usarla per iniziare a decodificare la prossima istruzione, mentre l'istruzione precedente non è stata ancora completata.

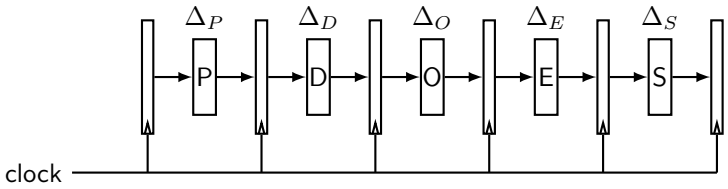

Figura 1: Esempio di pipeline

Più in generale, ci saranno altre reti logiche specializzate per compiti particolari, come accedere alla memoria o ai registri. Mentre una istruzione sta attraversando una certa fase del suo tragitto all'interno della CPU, sta occupando una sola particolare rete logica, mentre le altre sono al momento inutilizzate, e potrebbero essere utilizzate per svolgere il loro compito su altre istruzioni. Nel caso ideale sarà possibile identificare una serie di fasi attraverso cui tutte le istruzioni devono passare e tali che ogni fase usa una rete indipendente dalle altre. Nella metafora della catena di montaggio, ciascuna di queste reti è un operaio specializzato nel suo compito, gli operai sono in fila davanti al nastro, e il nastro trasporta la sequenza di istruzioni da elaborare. Per fissare le idee, supponiamo che le fasi siano quelle dell'esempio precedente: prelievo (P), decodifica (D), prelievo operandi (O), esecuzione (E) e scrittura risultato (S). Lo stato di una pipeline viene in genere illustrato nel modo seguente:

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
i	P	D	O	E	S				
$i+1$		P	D	O	E	S			
$i+2$			P	D	O	E	S		
$i+3$				P	D	O	E	S	
$i+4$					P	D	O	E	S

Ogni riga della tabella mostra l'elaborazione di una singola istruzione nel tempo, mentre ogni colonna mostra lo stato della pipeline ad un certo periodo di clock. Nel periodo t_0 la pipeline contiene solo l'istruzione i, nello stadio P. Nel periodo successivo (t_1) l'istruzione i si sposta nello stadio D, liberando lo stadio P che viene occupato dall'istruzione successiva, $i+1$. Nel periodo t_4 tutti gli stadi sono occupati, ciascuno da una istruzione diversa. Alla fine del periodo t_4 l'istruzione i viene completata, seguita dall'istruzione $i+1$ alla fine del periodo t_5, e così via.

Per realizzare la pipeline possiamo organizzare il processore come mostrato in Figura 1. Nella Figura, i blocchi P, D, O, E e S rappresentano le reti combinatorie che svolgono il corrispondente compito. Tra ogni coppia di reti inseriamo dei *registri di pipeline* che alla fine di ogni periodo di clock registrano il risultato dello stadio della pipeline che li precede e, all'inizio del clock successivo, lo rendono disponbile allo stadio della pipeline che li segue. Il periodo di clock Δ deve essere scelto in modo che sia maggiore del tempo di attraversamento di tutte le

reti combinatorie (più il tempo di setup dei registri, che assumiamo uguale per tutti i registri):

$$\Delta = \max\{\, \Delta_P, \Delta_D, \Delta_O, \Delta_E, \Delta_S \,\} + t_{setup}.$$

Cosa guadagniamo rispetto ad un processore che faccia le stesse cose, ma senza pipeline? Se il processore senza pipeline esegue le varie fasi (P, D, O, E, S) in cicli di clock diversi, completa una istruzione ogni 5 periodi di clock, mentre il processore con pipeline (nel caso ideale) ne completa una ogni ciclo di clock, quindi il numero di istruzioni per unità di tempo aumenta di 5 volte, pari al numero di stadi della pipeline (detto anche *profondità della pipeline*). Se il processore senza pipeline esegue tutte le azioni in un unico periodo di clock, il suo periodo di clock dove essere almeno pari alla *somma* $\Delta_P+\Delta_D+\Delta_O+\Delta_E+\Delta_S$. In questo caso la pipeline ci permette di aumentare la frequenza di clock, ottenendo anche in questo caso un aumento del numero di istruzioni completate per unità di tempo, con un fattore praticamente uguale al numero di stadi della pipeline (purché tutti gli stati abbiano un tempo di attraversamento all'incirca uguale).

1 Le alee

Ovviamente il caso in cui si riesce a completare una nuova istruzione ad ogni ciclo di clock è solo il caso ideale: in pratica ci sono varie condizioni che impediscono di tenere questo ritmo in modo costante. Un primo problema è dato dalle *àlee*, situazioni che, se non correttamente gestite, possono portare ad una esecuzione errata del programma rispetto a come sarebbe stato eseguito su un processore senza pipeline. Le alee si distinguono in:

- alee strutturali;
- alee sui dati;
- alee sul controllo.

1.1 Alee strutturali

Le alee strutturali si verificano quando due o più istruzioni che si trovano in due stati diversi della pipeline cercano di accedere contemporaneamente alla stessa risorsa. Per esempio, una istruzione i con un operando sorgente in memoria si trova nello stadio O, e un'altra istruzione j con operando destinazione in memoria si trova nello stadio S: entrambe cercherebbero di accedere contemporaneamente alla memoria. Le alee strutturali si possono risolvere introducendo degli "stalli" o "bolle" nella pipeline: nel caso di esempio, si ferma l'istruzione i nello stadio D fino a quando l'istruzione j non è uscita dallo stadio S. Si noti che mentre i è ferma nello stadio D, anche lo stadio precedente, P, deve fermarsi. In generale, data la natura strettamente sequenziale della pipeline, se una istruzione è ferma in un qualche stadio, anche tutti gli stadi precedenti si devono fermare. Nel periodo di clock in cui i sarebbe dovuta entrare nello

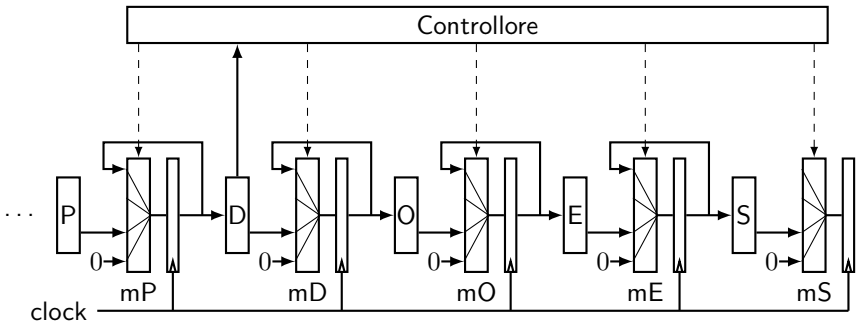

Figura 2: Circuito di controllo della pipeline

stadio O, lo stadio O sarà occupato da una cosiddetta "bolla"—essenzialmente una `nop`.

Operativamente, possiamo organizzare la pipeline come in Figura 2, dove abbiamo inserito un multiplexer di fronte ad ogni registro di pipeline, in modo che ogni registro possa ricevere o l'uscita dello stadio precedente, o la sua stessa uscita dal periodo precedente, o una costante che codifica la bolla (rappresentata con 0 in Figura). Aggiungiamo poi un controllore che tiene traccia di tutte le istruzioni che entrano nella pipeline (una volta decodificate) e, per ogni periodo, decide quale via attivare in ogni multiplexer (nella Figura abbiamo omesso per il momento il circuito che deve precede lo stadio P e decide da quale indirizzo prelevare la prossima istruzione). Per esempio, supponiamo che al clock t lo stadio E contenga una istruzione i che dovrà scrivere in memoria quando raggiunge lo stadio S, e che lo stadio D abbia decodificato una istruzione j che dovrà usare la memoria quando raggiunge lo stadio O. Il controllore riconosce l'alea strutturale e, per il periodo di clock $t + 1$, seleziona la via in alto del multiplexer mP (mantieni lo stato corrente), la via in basso del multiplxer mD (inserisci una bolla) e le vie normali dei restanti multiplexer. Durante il clock $t + 1$ l'istruzione j passa nello stadio S, dove esegue la sua scrittura in memoria indisturbata, mentre l'istruzione i è ferma nello stadio D. Per il clock $t + 2$ il controllore selezionerà tutte le vie normali, e l'istruzione i potrà riprendere il suo cammino.

Una volta introdotta nella pipeline, la bolla deve poi attraversare tutti gli stadi fino all'uscita: ogni bolla introdotta riduce quindi il numero di istruzioni completate per unità di tempo, e i progettisti delle pipeline cercano sempre di ridurle al minimo. Nel caso delle alee strutturali, una strategia che permette di ridurre le bolle/stalli è quella di aumentare le risorse hardware. Per esempio, nella pipeline di Figura 1 abbiamo un alea strutturale con tutte le istruzioni che accedono in memoria, in quanto lo stadio P deve sempre accedere anch'esso alla memoria per prelevare altre istruzioni. Questa alea può essere risolta introducendo cache separate per le istruzioni e per i dati.

Un'altra strategia molto efficace consiste nel progettare il set di istruzioni direttamente in funzione della sua elaborazione tramite una pipeline. In partico-

lare, ogni istruzione dovrebbe fare poche cose semplici, affidando al compilatore il compito di codificare le operazioni più elaborate usando più istruzioni. I processori progettati in questo modo sono chiamati RISC, per Reduced Instruction Set Computer, e i loro ideatori li contrapposero ai processori CISC, per Complex Instruction Set Computer. I processori RISC sono nati nelle università negli anni '80 del secolo scorso; la loro adozione non è stata immediata, e anzi ne nacque una lunga polemica, ma alla fine l'architettura RISC è diventata quella prevalente al giorno d'oggi (anche nel caso dell'Intel, come vedremo).

Un set di istruzioni di tipo RISC prevede, tipicamente:

- istruzioni tutte della stessa grandezza (per es. 4 byte);
- pochi e semplici formati;
- istruzioni operative che lavorano esclusivamente sui registri, o al massimo su operandi immediati;
- istruzioni separate di load/store per accedere alla memoria, senza fare altre elaborazioni.

La tipica istruzione operativa RISC ha il formato "*op* src_1, src_2, *dst*", dove *op* è il codice operativo e src_1, src_2 e *dst* sono due registri sorgenti e una destinazione. Una istruzione di load può avere il formato "`ld` *offset* `(`*base*`)`, *dst*" e una store "`st` *src*, *offset* `(`*base*`)`", dove *src*, *dst* e *base* sono registri. Le istruzioni di load e store completano il loro accesso in memoria nello stadio E, e dunque non entrano mai in conflitto tra loro e con le altre istruzioni. Nello stadio O vengono solo letti registri e nello stadio S solo scritti registri, e le due operazioni possono essere svolte contemporaneamente nello stesso periodo di clock.

Il fatto che le istruzioni abbiano tutte la stessa grandezza semplifica lo stadio P, che può autonomamente calcolare l'indirizzo della prossima istruzione da prelevare, senza aspettare che questa venga decodificata (ovviamente se non si tratta di un salto, si vedano le alee sul controllo più avanti).

1.2 Alee sui dati

Le alee sui dati si presentano quando una istruzione potrebbe leggere un dato diverso rispetto a quello che avrebbe letto in un processore senza pipeline.

Consideriamo per esempio la sequenza di istruzioni (RISC)

```
add r1, r2, r3
sub r3, r4, r5
```

L'istruzione `sub` legge il registro `r3`, che in un processore senza pipeline dovrebbe contere il risultato della `add` precedente. Nella pipeline, però, il registro `r3` verrà aggiornato alla fine dello stadio S, mentre l'istruzione `sub` ne ha bisogno all'inizio dello stadio O. Quando l'istruzione `sub` si trova nello stadio O, l'istruzione `add` si trova ancora nello stadio E, e devono passare ancora due cicli di clock prima che `r3` venga aggiornato. Se non si prendessero provvedimenti, la `sub` leggerebbe il *vecchio* contenuto di `r3`, non il valore calcolato dalla `add`.

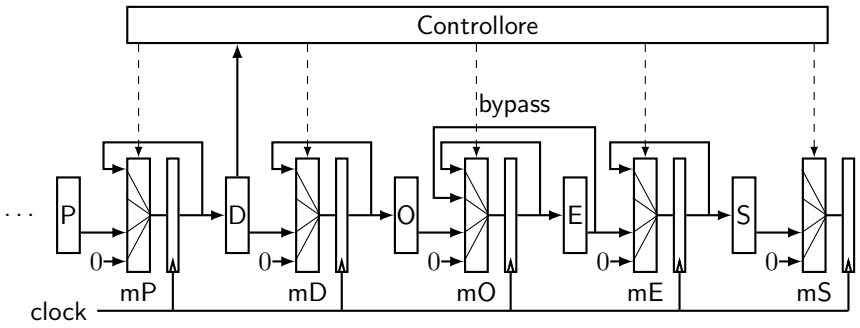

Figura 3: Circuito di bypass

Anche le alee sui dati si possono risolvere introducendo bolle. In questo caso il controllore dovrebbe tenere ferma la `sub` per due cicli nello stadio D, introducendo due bolle. In generale, tutte le alee possono risolversi introducendo bolle: al limite, se inseriamo 4 bolle dopo ogni istruzione, siamo ritornati al processore senza pipeline, che sicuramente funziona. Ovviamente, però, perderemmo tutti i vantaggi in termini di prestazioni.

Per risolvere le alee sui dati senza introdurre bolle possiamo aggiungere un circuito di *bypass* come in Figura 3. L'idea è che le istruzioni non hanno veramente bisogno di eseguire una lettura da un particolare registro, ma solo di ricevere il valore calcolato da una precedente istruzione. Nell'esempio precedente, la `sub` ha solo bisogno del risultato della `add`, e questo è già disponibile alla fine del ciclo di clock in cui la `add` si trova nello stadio E. Il controllore può a questo punto selezionare la via del bypass nel multiplexer mO, in modo che, al clock successivo, quando la `sub` si trova nello stadio E, riceva il risultato della precedente `add` al posto del valore letto dallo stadio O (il circuito in Figura è puramente indicativo: il circuito completo deve tenere conto del fatto che un operando arriva dal bypass e l'altro dallo stadio O).

1.3 Alee sul controllo

Le alee sul controllo si verificano quando il processo con pipeline potrebbe eseguire istruzioni diverse da quelle eseguite dal processore senza pipeline. Questo può succedere quando viene prelevata una istruzione di salto condizionale, o una istruzione di salto indiretto. In entrambi i casi l'indirizzo dell'istruzione successiva diventa noto solo uando l'istruzione di salto si trova più avanti nella pipeline, mentre lo stadio P ha bisogno di saperlo subito.

Anche qui il controllore della pipeline può risolvere il problema introducendo un numero sufficiente di bolle, ma ancora una volta si può provare a fare di meglio. In questo caso, il processore deve tentare di indovinare quale sarà la prossima istruzione, prima di eseguire completamente l'istruzione di salto, e continuare a prelevare istruzioni dall'indirizzo "predetto". Dopo alcuni cicli di clock, l'istruzione di salto sarà arrivata allo stadio in cui viene calcolata la

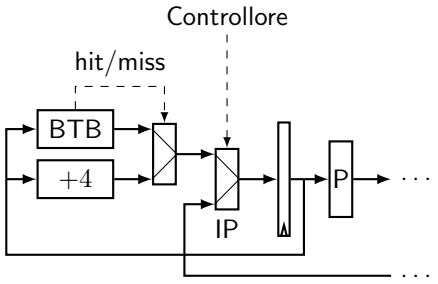

Figura 4: Branch Target Buffer

vera destinazione (per esempio nello stadio E). A quel punto il controllore può confrontare la vera destinazione con quella che era stata predetta: se aveva indovinato va tutto bene e le istruzioni prelevate nel frattempo possono proseguire attraverso la pipeline; se aveva sbagliato, per fortuna nessuna delle istruzioni prelevate ha ancora avuto effetti permamenti (tipo scritture in memoria o nei registri, che avvengono solo negli stadi E ed S), quindi possono essere tutte annullate (sostanzialmente trasformandole in bolle) e lo stadio P può ripartire a prelevare dall'indirizzo corretto. La bontà di questa strategia dipende da quanto efficacemente si riesce a prevedere le destinazioni dei salti, e molte risorse sono state investite nel tentativo di migliorare questo aspetto.

La strategia più semplice è di fare una previsione statica. Per esempio, se il salto è all'indietro, si può assumere che il programma contenga un ciclo e che questo ciclo venga effettivamente eseguito un po' di volte, quindi si può predire che il salto verrà fatto. Viceversa, se il salto è in avanti, si può assumere che il programmatore (o il compilatore) abbia organizzato il codice in modo che il caso più frequente sia quello in cui il salto non viene fatto, e dunque si può prevedere che non verrà fatto.

Strategie più sofisticate prevedono di ricordare in un nuovo dispositivo di cache, detto Branch Target Buffer (BTB), informazioni sui salti che sono stati osservati in passato, e basare la previsione su queste informazioni. La Figura 4 mostra una schematizzazione della parte di pipeline che pilota lo stadio di prelievo. Lo stadio P preleva l'istruzione all'indirizzo contenuto nell'Instruction Pointer (`rip` nei processori Intel). Il controllore decide se aggiornare l'instruction pointer con il valore predetto (via alta del secondo multiplexer) o il valore calcolato dalla pipeline (via bassa). Il valore predetto è ottenuto o dal valore precedente incrementato di 4 (ipotizzando istruzioni RISC) o, se il valore precedente è stato trovato nel BTB, dal valore predetto dal BTB stesso.

Una strategia molto semplice per le previsioni del BTB è di ricordare soltanto cosa l'istruzione aveva fatto l'ultima volta che era stata eseguita, e rifare la stessa cosa. Questa strategia non è molto efficace, perché in presenza di un ciclo sbaglia più del necessario: sbaglia sicuramente quando si esce dal ciclo, ma sbaglia di nuovo anche se il programma ripassa in seguito dallo stesso ciclo (perché ricorda che l'ultima volta il salto non era stato fatto).

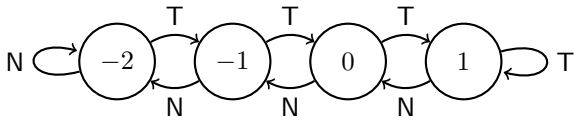

Figura 5: Predittore di salto (2 bit con saturazione).

Un metodo più efficace, che è anche quello comunemente usato, è di usare due bit organizzati come un contatore con saturazione. L'idea è illustrata in Figura 5: il contatore è inizialmente a zero e, ogni volta che il salto viene eseguito (rami T per *Taken*) il contatore viene aumentanto (saturando a 1); ogni volta che il salto non viene eseguito (rami N per *Not taken*) il contatore viene decrementato. Il BTP predirrà che il salto non verrà fatto se il contatore è negativo, e fatto altrimenti. Nell'esempio precedente, il contatore arriverà presto a 1 mentre il ciclo viene eseguito e sbaglierà una volta all'uscita dal ciclo, ma questa volta riporterà il contatore a zero e, se lo stesso ciclo viene incontrato una seconda volta, predirrà correttamente che il salto verrà effettuato. I predittori moderni basano le loro previsioni sulla storia degli ultimi salti, memorizzata in degli shift-register in cui ogni bit rappresenta l'esito di un salto.

Notare che le istruzioni `ret` sono istruzioni di salto indiretto, e per sapere a quale indirizzo devono essere prelevate le prossime istruzioni occorre eseguire una operazione di lettura in memoria. Le istruzioni `ret` possono però essere trattate a parte, perché nei casi normali sono strettamente associate alle `call`. Molti processori (inclusi gli Intel) hanno un Return Address Stack (RAS), che è un buffer interno al processore, organizzato a pila. Ogni istruzione `call` inserisce l'indirizzo di ritorno in cima alla pila e ogni `ret` estrae dalla cima della pila. Notare che si tratta comunqe di una previsione, sia perché i programmi sono liberi di usare le `ret` anche in modi non legati alle `call`, sia perché il RAS ha comunque una dimensione limitata e ricorda solo un piccolo numero degli ultimi indirizzi di ritorno.

2 La pipeline nei processori Intel

Una delle operazioni più complesse nell'elaborazione del set di istruzioni Intel è la decodifica: le istruzioni hanno lunghezza variabile e la codifica ha tanti casi particolari da gestire. Per esempio, una istruzione come "`push` *registro*" è codificata su un solo byte con valore esadecimale `50` più il codice del registro, secondo la corrispondenza `rax` = 0, `rcx` = 1, `rdx` = 2, `rbx` = 3, `rsp` = 4, `rbp` = 5, `rsi` = 6 e `rdi` = 8. Quindi, per esempio, "`push rbp`" è codificata con `55`. Per usare i registri (o le parti di registro) aggiunti da AMD nel processore a 64 bit bisogna usare il cosiddetto prefisso REX, che ha un valore esadecimale da `40` a `4F`[1]. Per esempio, l'istruzione "`push r9`" si codifica su due byte, `41 51`.

[1] Nel processore a 32 bit questi byte codificavano le istruzioni "`inc` *resistro*" e "`dec` *registro*" su un byte; nel processore a 64 bit queste istruzioni devono essere codificate su più byte.

Il prefisso REX va utilizzato anche se si vogliono usare operandi a 64 bit nelle istruzioni i cui operandi sono per default a 32 bit (praticamente tutte tranne le `push`, `pop` e poche altre). Quasi tutte le istruzioni che specificano due operandi hanno un codice operativo (su 1, 2 o 3 byte) e devono usare un ulteriore byte, detto ModR/M, che permette di specificare un registro (secondo la codifica di cui sopra, su 3 bit) e un altro operando che può essere a sua volta in un registro o in memoria. Il byte ModR/M permette di codificare gli indirizzamenti diretti, indiretti tramite registro, e gli indirizzamenti con base più offset; l'offset può essere codificato su 1, 2 o 4 byte e deve seguire il byte ModR/M. Se si utilizza un indirizzamento con registro indice (con o senza scala), è necessario un ulteriore byte, detto SIB (Scale Index Base). Se l'istruzione ha un operando immediato, anche questo può essere codificato su 1, 2 o 4 byte e si trova alla fine dell'istruzione. A tutto ciò aggiungiamo il fatto che l'istruzione può essere preceduta da uno o più byte di prefisso (come `rep`).

Per esempio, vediamo cosa significano i byte `48 01 44 cb 10`, che codificano l'istruzione "`add %rax, 16(%rbx, %rcx, 8)`". Il byte `48` è un prefisso REX, necessario perché l'istruzione usa operandi a 64 bit e la `add`, per default, ha operandi a 32 bit. Il byte `01` è il codice operativo della `add`, che specifica anche l'ordine degli operandi (la stessa `add` a operandi invertiti ha codice operativo `03` e tutto il resto uguale); segue il byte ModR/M che ha tre campi. Scritti in binario e partendo dal bit più significativo, i campi sono: Mod = 01, reg = 000 e r/m = 100. Il campo reg indica il registro `rax`, mentre la combinazione Mod = 01 e r/m = 100 indica la presenza di un byte SIB e di un offset codificato su un byte; il byte SIB è `cb` e l'offset è `10` (pari a 16); anche il byte SIB ha tre campi, che scritti in binario partendo dal bit più significativo sono: S = 11, I = 001 e B = 011, vale a dire S = 3, I = 1 e B = 3. I tre campi codificano una scala di 8 (2^S), il registro indice (I) `rcx` e il registro base (B) `rbx`.

Anche se i processori Intel hanno un set di istruzioni molto complesso e non pensato per la realizzazione di una pipeline, Intel riuscì ad introdurre una pipeline a 5 stadi nel processore 80486 del 1989. All'epoca il processore era ancora a 32 bit e mancavano molte delle estensioni aggiunte in seguito, come le istruzioni SSE. Lo schema con codice operativo, ModR/M e SIB era però già presente. Per gestirne la complessità, lo stadio di fetch della pipeline del 486 preleva sempre 16 byte allineati naturalmente, senza sapere dove iniziano e dove finiscono le istruzioni al suo interno. Cerca sempre di mantenere un buffer di due di questi blocchi da 16 byte consecutivi, perché una istruzione potrebbe trovarsi anche a cavallo di un confine di 16 byte. Lo stadio di decodifica è scomposto in due sottostadi, D1 e D2. D1 decodifica al più 3 byte dell'istruzione e fornisce allo stadio D2 informazioni su come procedere con il resto della decodifica.

Una ulteriore complicazione è data dal fatto che le istruzioni Intel richiedono tempi molto diversi tra loro: una `add` tra registri può essere completata in un clock dello stadio di esecuzione, ma una `add` come quella esaminata sopra, che deve leggere e poi anche riscrivere in memoria, richiede 3 clock nello stadio di esecuzione. Il circuito di controllo della pipeline del 486 deve quindi poter mantenere una istruzione nello stesso stadio anche per più di un clock, mettendo

in stallo le istruzioni precedenti. Questo accade anche nello stadio D2 della decodifica, se l'istruzione contiene offset o operandi immediati.

Pur con tutte queste limitazioni e complicazioni, il 486 comunque riuscì ad andare circa al doppio della velocità del suo predecessore (80386) anche a parità di frequenza di clock, e riuscì anche ad essere competitivo con i processori RISC dell'epoca.