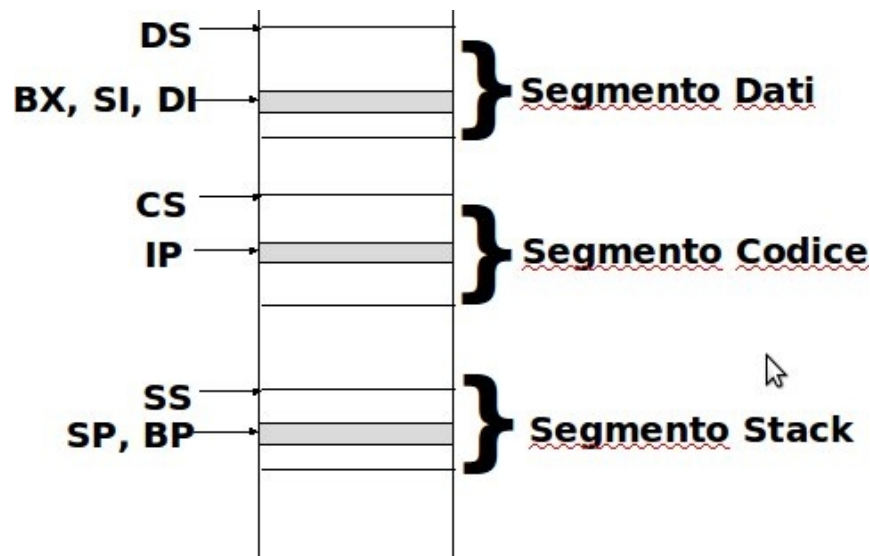


Segmentazione della memoria in Real Mode



Il processore è un modello a memoria segmentata non protetta. La segmentazione è visibile dal programmatore (o dal programma), mentre la paginazione è un meccanismo più nascosto. La segmentazione è stata introdotta per razionalizzare le protezioni e le articolazioni dei programmi a livello compilatore, per separare i moduli di codice e di dati dai moduli di stack. Lavora sulla memoria logica (cioè la memoria vista dalle istruzioni). La memoria può essere considerata come divisa in segmenti di 64 KB che iniziano con indirizzi multipli di 16. I gruppi di 16 byte che iniziano con indirizzi multipli di 16 vengono detti paragrafi. La gestione della segmentazione in RealMode è diversa da quella in ProtectedMode.

Calcolo degli indirizzi

Ogni volta che l'8086 deve generare un indirizzo da mettere sull'address bus, esso esegue un'operazione di somma tra il contenuto di registro puntatore oppure di BX (effective address o offset) e il contenuto di un registro di segmento (segment address). Questa somma avviene dopo aver moltiplicato per 16 il valore del registro di segmento.

Il procedimento può essere così riassunto:

Effective Address	16 bit	+	
Segment Address	16 bit	+	0000 =
Physical Address	20 bit		

L'assemblatore lascia libertà se omettere l'indirizzo di segmento oppure dichiararlo esplicitamente (es DS:[BX]). Di base però l'assemblatore lascia di default la possibilità di ometterlo e in automatico andrà a indirizzarlo.

Essendo ogni registro di segmento e di offset da 16 bit ciascuno sembra che l'indirizzo sia di 32 bit. In realtà, l'indirizzo effettivo è di 16 bit perché si utilizza uno schema di memoria a blocchi sovrapponibili in cui lo spazio non sovrapponibile è di 16 byte.

Modelli di memoria

Il modello della memoria logica in Modo Reale ha una profondità di 1 MB, quindi l'indirizzamento fisica deve essere espresso su 20 bit, ed è organizzata in blocchi da 64 kB. In questa modalità i segmenti sono sovrapposti, mentre nella modalità Protected Mode si presentano disgiunti.

Esistono 3 modelli di memoria:

- Flat memory

La memoria appare come un singolo blocco (Linear Address Space) di dimensioni $(2^{32}-1)$ Byte.

- Segmented Memory Model (usata in modo protetto)

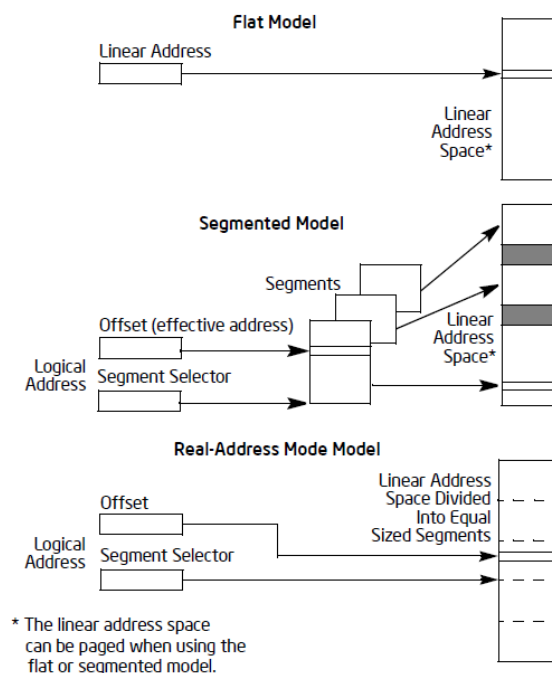
La memoria appare come un gruppo di spazi d'indirizzamento indipendenti, chiamati segmenti. Ogni applicazione può indirizzare fino a $(2^{16}-1)$ segmenti di differenti dimensioni e tipologie.

Ogni segmento può essere di dimensioni massime 232 Byte.

Il programma, per accedere ad un Byte, si riferisce ad un indirizzo logico. Esso è definito mediante un Segment Selector e un Offset.

- Real-address mode Memory Model (usata in modo reale)

E' il modello di memoria per i processori Intel 8086 compatibili. Andremo ad analizzare in modo approfondito solo quest'ultima modalità.

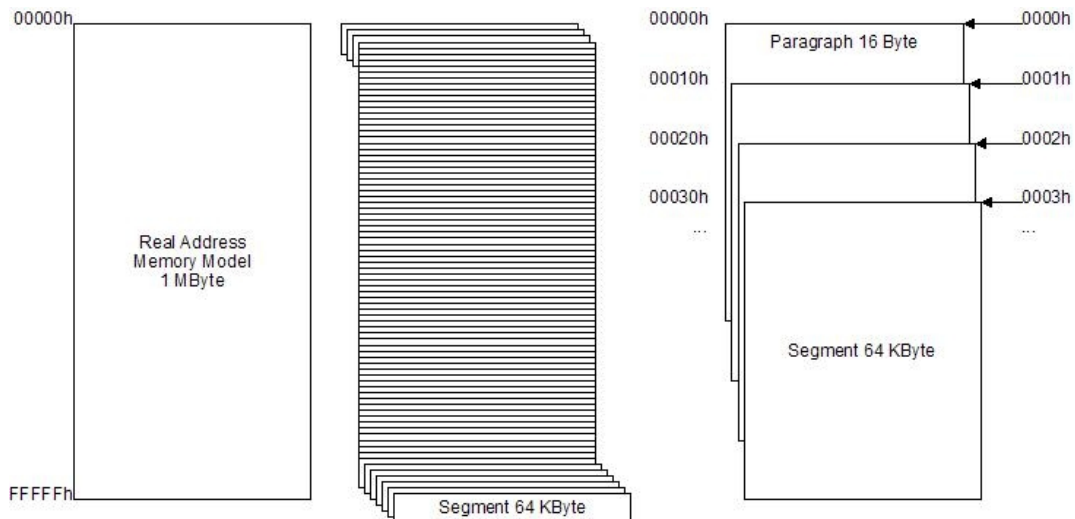


Real-address mode Memory Model

I segmenti sono visibili al programmatore a livello istruzione.

Essi permettono di organizzare la memoria dividendola in porzioni omogenee (dati, codice e stack), anche più di una per tipo.

I segmenti sono da 64 KByte, in overlapping di un paragrafo (16 Byte) ognuno.



Tale tipo di segmentazione minimizza lo spreco di memoria. Inoltre, si ottiene uno spazio di allocazione (virtuale) a 20bit chiuso su se stesso.

Ad esempio, posizionandoci all'ultimo segmento (che fisicamente sarà di lunghezza pari a 16Byte)

Segment Address FFFFh

E imponendo un offset superiore a 16 Byte

Offset 0011h

Ci stiamo riferendo all'indirizzo

Physical Address 00001h

Fisicamente, l'indirizzo punta al 2° Byte dello spazio di indirizzamento.

In alcune architetture non è possibile manipolare i registri di segmento, ma solo gli offset.

In questi casi è il Loader dell'O.S. ad occuparsi del caricamento di tali registri.

Organizzazione della memoria

L'organizzazione della memoria è caratterizzata da limitazioni hardware definite dal progettista.

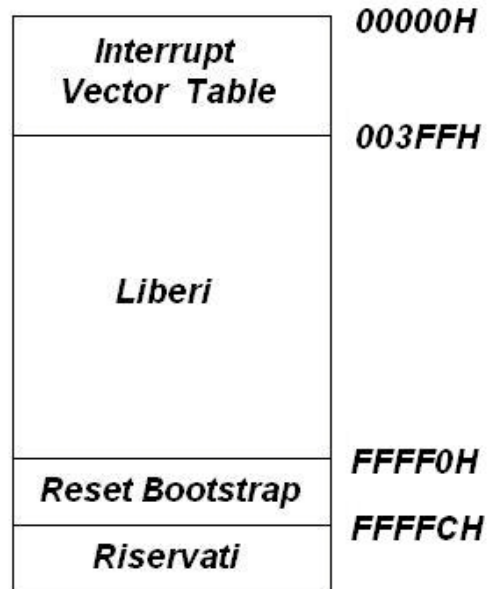
Il PROGRAM COUNTER (PC), o INSTRUCTION POINTER (IP), è un registro della CPU la cui funzione è quella di conservare l'indirizzo in memoria dell'istruzione successiva da eseguire.

Il PC, per esempio all'avvio della macchina, deve essere inizializzato ad un valore cui corrisponde la prima istruzione da eseguire.

La scelta è normalmente quella di definire l'indirizzo di partenza della prima istruzione o in testa o in coda nella memoria.

La scelta della piattaforma x86 (a partire dalla versione 8086 fino ad arrivare ai dual-core e ai quad-core) è di partire con l'istruzione dal fondo della memoria. Per esempio, l'8086 dopo la ricezione del segnale di reset si procura il codice della prima istruzione da eseguire all'indirizzo a 20 bit

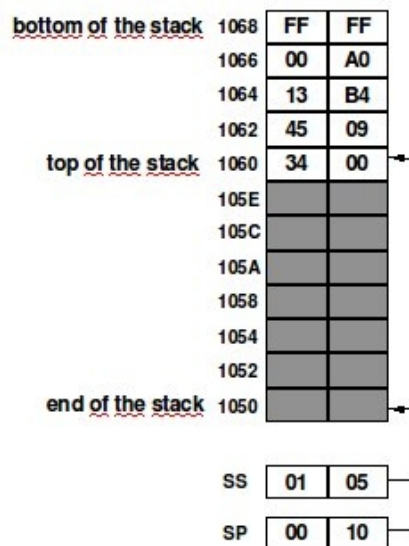
FFFF0H, cioè 16 byte prima della fine della sua memoria. Nel Pentium, con ABUS a 32 bit, l'indirizzo iniziale sarà FFFFFFFF0. A quella locazione deve essere presente una memoria elettronica non volatile, ROM o EPROM. La prima istruzione farà saltare (JMP) ad un piccolo programma di test e di caricamento residente in ROM detto "POST" (Power On Self Test). Alcuni parti di memoria quindi sono dedicate:



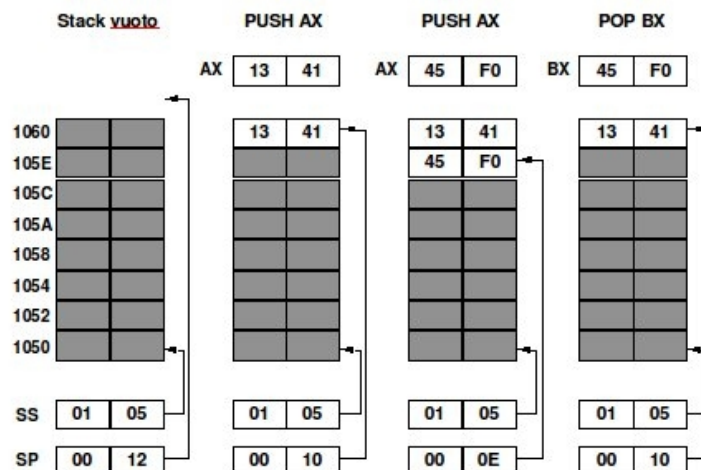
Alcune parti della memoria non sono libere, ma riservate. Ad esempio la memoria dall'indirizzo 00000H allo 003FFH è riservata, in quanto contiene la *Interrupt Vector Table*. Gli indirizzi da FFFF0H a FFFFBH sono utilizzati per contenere un'istruzione di salto alla routine di caricamento di programma di bootstrap. Le locazioni da FFFFCH a FFFFFFH sono invece riservate dati di versione del BIOS.

Lo Stack

L'80x86 prevede alcune strutture e meccanismi hardware per la gestione di uno *stack*. Lo stack corrisponde al segmento di memoria la cui testa è puntata da SS. Il *top* dello stack (locazione riempita per ultima) è puntato da SP (o ESP, in IA-32). Lo stack cresce dalle locazioni di memoria con indirizzo maggiore verso quelle ad indirizzo minore. Ad ogni operazione di PUSH decrementa di 2 unità SP e scrive una word nella locazione da questo puntata, mentre ad ogni operazione di POP estrae una word dalla locazione puntata da SP, e successivamente incrementa SP di 2 unità.



Esempio di operazioni sullo stack:



Pipeline

Come accennato nelle prime lezioni, uno dei metodi per aumentare le prestazioni di un sistema basato su microprocessore è quello di sfruttare l'esecuzione delle singole istruzioni (ILP) mediante una *pipeline*.

In informatica col termine pipeline si indica un processo di esecuzione (ad esempio l'esecuzione di un'istruzione) realizzato in più fasi, dette stadi.. Ogni fase, fisicamente realizzata in un blocco circuitale, provvede a ricevere in ingresso un dato, ad elaborarlo e poi a trasmetterlo all'elemento successivo. Il flusso di dato, nel nostro caso l'istruzione, percorre tutti gli elementi della pipeline.” I processori antecedenti al Pentium realizzavano la pipeline “imperfetta”; dal Pentium in poi si ha una pipeline (quasi) perfetta.

E' pertanto necessario suddividere le istruzioni in elementi atomici (non ulteriormente suddivisibili), assegnando ciascun elemento ad uno stadio.

Il principio di funzionamento di un pipeline è paragonabile a quello di una catena di montaggio.

L'assemblaggio è caratterizzato da diversi stadi da eseguire in un ordine fisso e da un determinato tempo di esecuzione. I vari stadi produttivi sono posti in cascata e lavorano in modo che l'uscita di uno stadio coincida con l'ingresso dello stadio seguente. A regime, ad ogni ciclo di clock viene terminata un'istruzione.

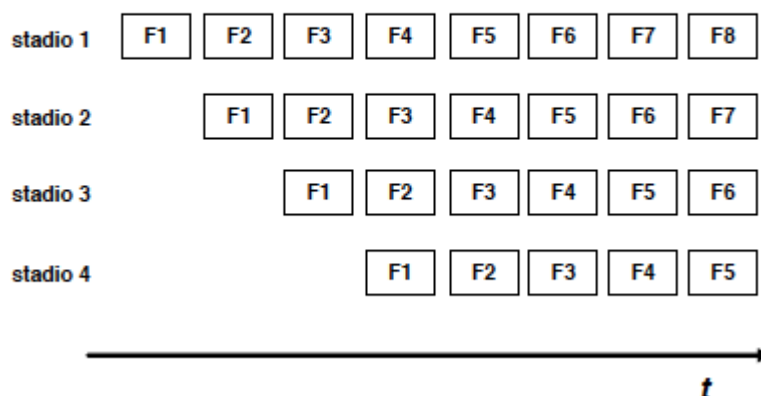
Si ha un pipeline ideale (o perfetto) quando la durata di esecuzione di tutte le fasi è identica (costante ed uguale). Se ciò non avviene si hanno dei periodi di non attività (pipeline imperfetto).



Nel caso in cui il tempo di stadio sia uguale in tutti gli stadi:

$$[\text{NumeroElementiPipeline}] \cdot [\text{TempoStadio}] = [\text{DurataIstruzione}]$$

Pipeline ideale



Nella progettazione di pipeline verso il pipeline perfetto si devono trovare gli stadi di pipeline per un'istruzione, vedere se sono omogenei e determinarne la durata massima. La frequenza di clock di riferimento è legata alla durata degli stadi di pipeline o più propriamente al maggiore periodo di tempo impiegato tra i diversi stadi. Più il tempo di un stadio è breve più il clock è veloce.

Il numero di stadi determina il throughput o meglio il numero di istruzioni eseguite al secondo.

Maggiori sono gli stadi di pipeline, più è difficile omogeneizzare la durata di tutti gli stadi

(ottimizzare la pipeline).

Sia detto:

- T_s il tempo di esecuzione di un'istruzione in un sistema senza pipeline
- T_p il tempo di esecuzione di un'istruzione in un sistema con pipeline

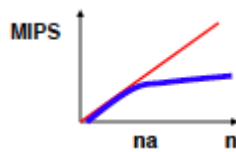
Si ha che $T_s \leq T_p$: infatti, il tempo di esecuzione di un singolo stadio è dimensionato al tempo massimo di esecuzione nello stadio tra tutte le istruzioni del processore e pertanto la durata di una singola istruzione in una architettura pipeline può risultare maggiore (avviene tra l'altro nel caso di pipeline imperfetto o nel caso di blocco del pipeline perfetto).

Se si considera il throughput, le prestazioni vanno valutate in MIPS:

- $MIPS_s = 1/T_s$ (sistema senza pipeline)
- $MIPS_p = n/T_p$ (sistema con pipeline), con n numero di stadi

Si ha che $MIPS_s \ll MIPS_p$.

In teoria, le prestazioni crescono linearmente al crescere di n . In realtà, per $n \geq n_a$ l'aumento di prestazioni è quasi nullo poiché al crescere di n non si riesce più a soddisfare il vincolo di pipeline ottimo.



Valutazione delle prestazioni

Siano:

- k = numero stadi pipeline
- τ_i = tempi dell' i -esimo stadio
- $\tau_{\max} = \max(\tau_i)$

Se consideriamo le N istruzioni, si ha:

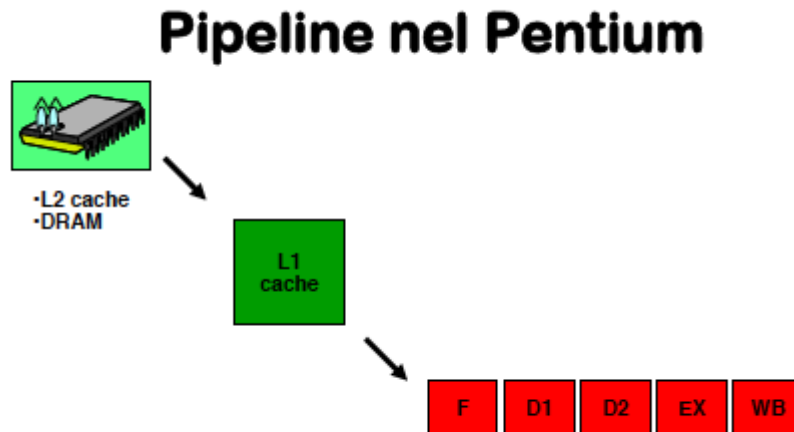
- Tempo di esecuzione di una istruzione: $k\tau_{\max}$
- Tempo di esecuzione di N istruzioni: $Nk\tau_{\max}$
- Tempo di esecuzione di N istruzioni con pipeline: $k\tau_{\max} + (N-1)\tau_{\max}$

$k\tau_{\max}$ è il tempo di esecuzione della prima istruzione che deve percorrere tutta la pipeline.

Si può definire il fattore di accelerazione come il rapporto tra il tempo di esecuzione di N istruzioni senza pipeline e il tempo di esecuzione di N istruzioni con pipeline.

$$FattAccelerazione = \frac{N \cdot k \cdot \tau_{\max}}{k \cdot \tau_{\max} + (N-1) \cdot \tau_{\max}} \quad \text{per } N \text{ grandi tende a } k. \quad \text{In MIPS} \rightarrow \frac{1}{\tau_{\max}}$$

Pipeline nel Pentium



Originariamente esistevano 5 stadi di pipeline, nelle successive versioni si passa da 12 a 30. In generale, si cerca di avere stadi con “densità di lavoro” simile e frequenza di avanzamento dettata da τ_{max} .

Gli stadi di pipeline sono:

- *Fetch*: Non è la fase di prelevamento dalla memoria in quanto il tempo di accesso a quest'ultima è molto elevato rispetto alla velocità del clock; l'istruzione viene prelevata dalla L1 cache o dalla coda di prefetch
- *Decoding*: Se lavoriamo in modo reale 2 stadi di decodifica risultano eccessivi, invece in modo protetto dobbiamo eseguire 2 cose:
 - passaggio da indirizzo virtuale (logico) ad indirizzo reale (calcolo degli indirizzi); devono essere esaminate le opportune tabelle (D2)
 - verificare l'eseguibilità e la legittimità (privilegi) dell'istruzione (D1)
- *Execution*: Vengono eseguite le operazioni della ALU e settati i flag del PSW (per l'FPU c'è un altro pipeline)
- *Write-Back*: risultati scritti o nel registro o nella L1 cache o in memoria

Ogni stadio richiede un ciclo di clock; in alcune condizioni in uno stadio si inseriscono più periodi di clock, come nel caso dell'accesso alla memoria. In altre situazioni, ad esempio salti condizionati, si può determinare uno stallo completo e la conseguente necessità di svuotare la pipeline.

Esempio:

ADD AX, [BX]

- F: Devo andare a prelevare l'istruzione puntata da CS:IP, mandare sulla BIU (Bus Interface Unit) interna questo indirizzo e capire se si trova in memoria, in coda o in cache. Concettualmente viene preso il contenuto di CS:IP ed inserito nell'IR (fetch di 2 byte)
- D1: Dato che qui siamo in modo reale ed è tutto legittimo c'è solo l'interpretazione del codice operativo.
- D2: Si procura il dato relativo a [DS:BX]; passa dall'indirizzo segmentato a quello da spedire nell'ABUS (calcolo dell'indirizzo) e deve fare il fetch del dato (2 byte di AX). Si può avere un'altra situazione di stallo legata ai MISS; quindi i MISS possono verificarsi

sia in F che in D2.

- **EX**: Inserisce il valore del dato nel registro temporaneo dell'ALU. I Pentium hanno un moltiplicatore hardware. Si possono verificare criticità (stallo di esecuzione).
- **WB**: In questo caso non ci sono possibili criticità in quanto andiamo a scrivere in AX. Deve essere calcolato anche il nuovo Program Counter.

Il tempo reale di questa istruzione è il tempo di permanenza lungo la fase di pipeline:

$$T_{ADD} = \tau_F + \tau_{D1} + \tau_{D2} + \tau_{EX} + \tau_{WB}$$

$$\text{Best case: } T_{ADD} = T_p + T_p + T_p + T_p + T_p = 5 \cdot T_p$$

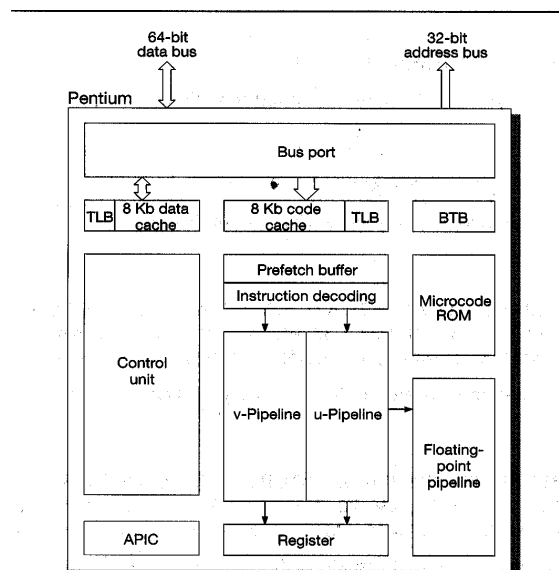
$$\text{Worst case: } T_{ADD} = m \cdot T_p + T_p + m \cdot T_p + T_p + T_p = 3 \cdot T_p + 2 \cdot m \cdot T_p = (3 + 2 \cdot m) \cdot T_p$$

dove m rappresenta il più piccolo multiplo intero di T_p che racchiude il tempo di accesso alla DRAM.

Processore superscalare

Un microprocessore con architettura superscalare supporta il calcolo parallelo su un singolo chip, permettendo prestazioni molto superiori a parità di clock rispetto ad una CPU ordinaria. Questa caratteristica è posseduta più o meno da tutte le CPU general purpose prodotte dal 1998.

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 (definite ALU). Le unità di controllo stabiliscono quali istruzioni possono essere eseguite in parallelo e le inviano alle rispettive unità. Questo compito non è facile, dato che un'istruzione può richiedere il risultato della precedente come proprio operando, oppure può dover impiegare il dato conservato in un registro usato anche dall'altra istruzione; il risultato può quindi cambiare secondo l'ordine d'esecuzione delle istruzioni. La maggior parte delle CPU moderne dedica molta potenza per svolgere questo compito con la massima precisione possibile, per permettere al processore di funzionare a pieno regime in modo costante; compito che si è reso sempre più importante con l'aumento del numero delle unità.

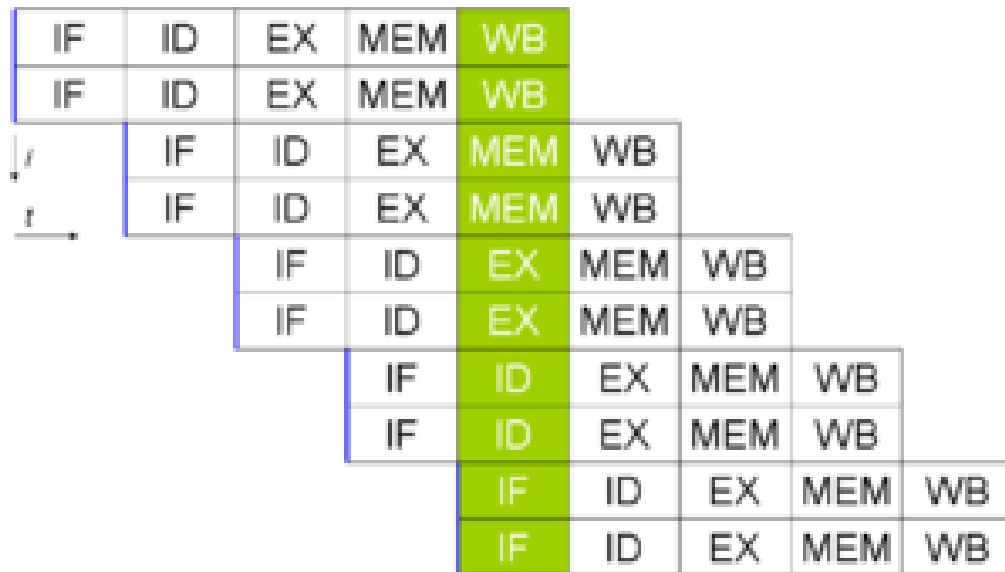


Se ad esempio caso avessimo 2 istruzioni, esse verranno veicolate separatamente ognuna su di una pipeline distinta.

Aumentando il numero di pipeline, però aumenta la complessità di valutare la compatibilità tra

istruzioni eseguite in pipeline diverse. Inoltre possono venirsi a formare delle situazioni di stallo nell'accesso a memorie lente oppure ad I/O. Anche i salti condizionati possono portare a situazioni critiche.

Esempio di pipeline ideale su due pipes:



Valutazione delle prestazioni

Nel caso di P pipeline operanti in parallelo

$$MIPS = (1/\tau_{max}) \cdot P = P \cdot \frac{(frequenza_{ClockCPU})}{m}$$

Essendo m il numero di periodi di clock richiesti per eseguire le operazioni in uno stadio un processore a i Ghz, con m=1, con 5 pipeline opera a

$$MIPS = 5 * 1000 \text{ (istruzioni/secondo)}$$