



Politecnico di Torino

III Facoltà di Ingegneria
Ingegneria Informatica

Wiki Book

Architetture dei Sistemi di Elaborazione Appunti Architettura 80x86

A cura di:

Gian Marco Pazzola e Giorgio Sogos (Concetti introduttivi e Architettura x86-32)

Pietro Russo (Bus di sistema)

Fabrizio Marongiu (Sottosistema di memoria nell'architettura 80x86)

ANNO ACCADEMICO 2010/2011

Indice

Indice	1
1. Concetti introduttivi.....	5
1.1 Le classi di computer	5
1.2 Prestazioni ed evoluzione.....	8
1.3 Architettura base dei microprocessori.....	14
1.4 Tecniche per incrementare le prestazioni.....	17
2. Architettura x86-32.....	25
2.1 Architettura Pentium 80x86, x64.....	25
2.2 Registri e modi	26
2.3 Gestione della memoria nel modo reale (segmentazione)	38
2.4 Organizzazione della memoria.....	47
2.4.1 Entry point	47
2.4.2 Interrupt vector table.....	48
2.4.3 Riservati	50
2.5 Stack.....	50
2.6 Architettura interna semplificata.....	53
2.7 Pipeline.....	56
2.7.1 Tempo di esecuzione e throughput	58
2.7.2 Valutazione delle prestazioni	59
2.7.3 Pipeline nel Pentium	61
2.7.4 Gestione della memoria	63

2.8 Processore superscalare.....	65
2.8.1 Valutazione delle prestazioni.....	67
2.8.2 Architettura superscalare di ultima generazione.....	68
2.9 I/O	69
2.9.1 Memory mapped I/O.....	69
2.9.2 Isolated I/O.....	70
2.9.3 Istruzioni di Input e Output.....	70
2.9.4 Vantaggi e Svantaggi	71
2.10 Registri speciali.....	72
2.10.1 FPU	72
2.10.1.1 Registri dati.....	73
2.10.1.2 Registri stato	74
2.10.1.3 Registri di controllo	75
2.10.3 SSE (Streaming SIMD Extensions).....	77
3. Bus di sistema	78
3.1 Definizioni generali.....	78
3.2 Cicli di Bus.....	80
3.3 Cicli di bus nel Pentium	84
3.4 Esempio di Pentium Bus Cycle con DRAM.....	91
3.5 PCI Bus	93
4. Sottosistema di memoria nell'architettura 80x86	96
4.1 Organizzazione fisica di una memoria: aspetti terminologici..	97

4.2 Organizzazione di una memoria	99
4.3 Banco di memoria SRAM.....	100
4.4 Breve cenno sull'interleaving	104
4.5 Le memorie DRAM	105
4.5.1 Ciclo di base di una DRAM.....	107
4.5.2 Ciclo di refresh.....	108
4.5.3 Fast Operative Mode.....	109
4.5.4 La famiglia delle DRAM	111
4.5.5 Temporizzazione delle SDRAM.....	112
4.5.6 DRAM Controller	115
4.5.7 Schema di memoria DRAM.....	116
4.6 Controllo d'errore	118
4.7 Esercizio relativo al progetto di una memoria	124
Riferimenti	126

1. Concetti introduttivi

1.1 Le classi di computer

In termini di aree di mercato si possono suddividere gli elaboratori in cinque diverse classi, ognuna di esse adatta nello svolgimento di particolari compiti. In generale, si possono distinguere le seguenti tipologie:

- Desktop / portatili
- Server
- Mainframe
- Supercomputer
- Embedded systems

La prima categoria *Desktop / portatili*, ricopre un'area di mercato abbastanza ampia che comprende sia pc di basse prestazioni che work station grafiche di prestazioni medio-alte. Inoltre è fondamentale il rapporto prezzo-prestazioni; difatti, quando si progettano questi sistemi l'architettura è funzione di un obiettivo: l'ottimizzazione di tale rapporto.

I costi possono variare da circa 500 € a 5.000 €.

La categoria dei *Server* possiede un'architettura pensata per ottimizzare particolari funzioni. Sono macchine che tipicamente non sono gestite a livello individuale ma sono collocate in data center dove sono interconnesse tra di loro mediante una “rete locale” e dove hanno la possibilità di comunicare con sistemi di storage mediante canali di connessione veloci quali la fibra ottica.

Queste macchine, inoltre, devono avere la possibilità di agire in un ambiente di virtualizzazione per garantire la scalabilità, parametro fondamentale di questi prodotti. In questi sistemi è necessario

avere un'elevata affidabilità (ridondanza di determinati componenti), scalabilità (possibilità di aumentare o diminuire il numero di macchine virtuali e/o le caratteristiche che i server hanno in modo statico e dinamico) ed un elevato throughput.

I costi variano dai 5.000 € a 4.000.000 €.

I *Mainframe* o sistemi centrali sono computer utilizzati per applicazioni critiche soprattutto da grandi aziende e istituzioni, tipicamente per elaborare con alte prestazioni ed alta affidabilità una grande mole di dati, come ad esempio quella in gioco nelle transazioni finanziarie, nei censimenti, nelle statistiche di industrie e clienti, nelle applicazioni ERP (Enterprise Resource Planning), ecc.

I moderni mainframe si distinguono dagli altri computer non tanto per la velocità di esecuzione di un singolo task (dipendente principalmente dalla frequenza del clock interno), ma piuttosto per le loro caratteristiche di disegno interno, che punta sull'alta affidabilità, sicurezza, bilanciamento delle prestazioni, e sulla compatibilità binaria dei programmi applicativi, caratteristica che garantisce nel tempo la protezione degli investimenti applicativi dei clienti. Questi elaboratori sono in grado di funzionare per anni senza interruzione e sono progettati in modo da consentire molte attività di riparazione e manutenzione senza richiedere il fermo della produzione.

I *Supercomputer* sono una categoria limitata ma economicamente rilevante. Esistono tanti supercomputer nel mondo; nel sito www.top500.org è possibile consultare una lista aggiornata dei supercomputer più potenti. Nel giugno 2010 la macchina più potente era un CRAY XT Jaguarad, situata negli USA, con una potenza di 2331 Tflops. Tale macchina è stata realizzata collegando decine di migliaia di microprocessori a 2.6 Ghz. Sono macchine costruite e cablate ad hoc, con un sistema operativo realizzato appositamente. I costi si aggirano attorno alle decine di milioni di euro

con una gestione particolarmente complessa. Al giorno d'oggi, queste macchine, hanno un ambito di utilizzo estremamente rilevante in alcuni contesti come quello farmaceutico. Le nuove medicine si provano principalmente sui supercomputer, aggregando e smontando modelli di molecole; questo è sicuramente più veloce rispetto che eseguire tali operazioni in un laboratorio.

La distinzione tra supercomputer e mainframe non è semplicissima, tuttavia i supercomputer generalmente si focalizzano su problemi che sono limitati dalla velocità di calcolo, mentre i mainframe si focalizzano su problemi che sono limitati dall'input/output e l'affidabilità.

Differenze e somiglianze includono:

- Entrambi i tipi di sistemi offrono la possibilità di calcolo parallelo. I supercomputer tipicamente la espongono al programmatore in maniera complessa, mentre i mainframe tipicamente la usano per eseguire molteplici task (multitasking). Un risultato di questa differenza è che aggiungendo processori al mainframe spesso accelera l'intero carico di lavoro in modo trasparente.
- I supercomputer sono ottimizzati per elaborazioni complesse che richiedono soprattutto grandi quantità di memoria, mentre i mainframe sono ottimizzati per elaborazioni relativamente più semplici che implicano però l'accesso rapido a grosse quantità di dati.
- I supercomputer sono spesso costruiti ad hoc per elaborazioni particolari. I mainframe invece vengono utilizzati per un'ampia gamma di elaborazioni ovvero sono molto più "general purpose". Di conseguenza, la maggior parte dei supercomputer vengono assemblati per esigenze specifiche e usi particolari, mentre i mainframe tipicamente formano una parte della linea di modelli di un produttore.
- I mainframe tendono ad avere un numero di processori di servizio che assistono i processori principali (per il supporto crittografico, la gestione dell'I/O, il monitoraggio, la gestione

della memoria, etc.) cosicché il numero effettivo dei processori presenti è molto elevato.

I sistemi *embedded* (*sistemi immersi o incorporati*) identificano genericamente tutti quei sistemi elettronici di elaborazione a microprocessore progettati appositamente per una determinata applicazione (*special purpose*) ovvero non riprogrammabili dall'utente per altri scopi. Tali sistemi sono realizzati su una piattaforma hardware ad hoc, integrati nel sistema che controllano ed in grado di gestire tutte o parte delle funzionalità. In questa area si collocano sistemi di svariate tipologie e dimensioni, in relazione al tipo di microprocessore, al sistema operativo, ed alla complessità del software che può variare da poche centinaia di byte a parecchi megabyte di codice. Appartengono a questa categoria di sistemi microelettronici di elaborazione i microcontrollori.

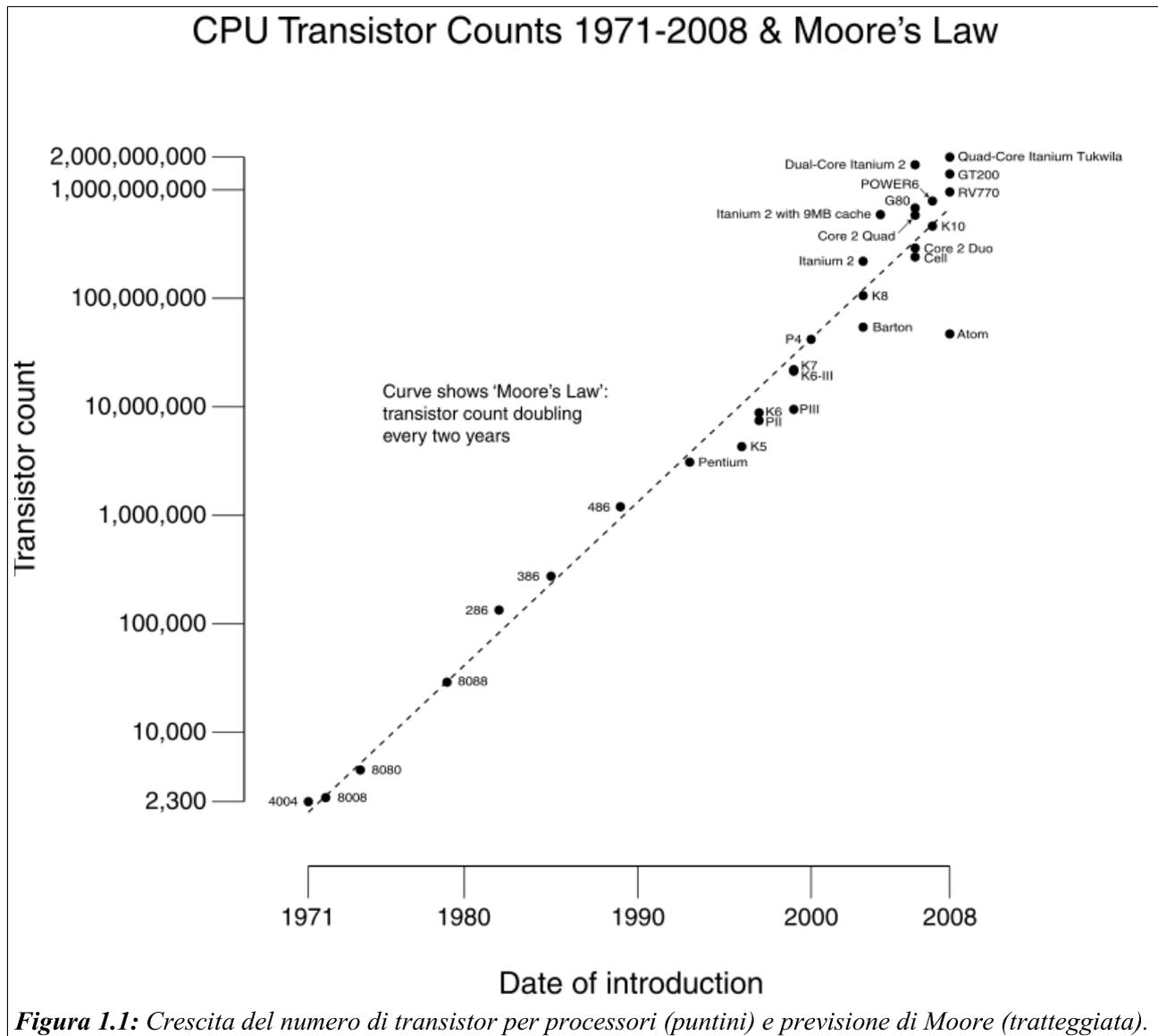
Contrariamente ai computer generici riprogrammabili (*general purpose*), un sistema *embedded* ha dei compiti noti già durante lo sviluppo, che eseguirà dunque grazie ad una combinazione hardware/software specificamente studiata per la tale applicazione. Grazie a ciò l'hardware può essere ridotto ai minimi termini per diminuire lo spazio occupato abbassando così anche i consumi, i tempi di elaborazione e il costo di fabbricazione. Inoltre l'esecuzione del software è spesso in tempo reale (*real-time*) per permettere un controllo deterministico dei tempi di esecuzione. Un esempio diffuso di Sistema *embedded* sono le centraline elettroniche installate a bordo degli autoveicoli per il controllo del motore e dell'ABS.

I costi possono variare dai 10 € ai 100.000 €.

1.2 Prestazioni ed evoluzione

Un aspetto fondamentale che ha permesso l'evoluzione e un conseguente aumento delle prestazioni è stato sicuramente l'aumento dei dispositivi (come transistor) a parità di superficie.

Legato a questo aspetto, nasce la legge di Moore che afferma come le prestazioni dei processori, e il numero di transistor ad esso relativo, raddoppino ogni 18 mesi.



Esempio:

Nel 1980, il microprocessore 8086 aveva circa 30.000 transistor e poteva eseguire circa un milione di operazioni al secondo.

Nel 1998, il Pentium IV aveva circa, a parità di superficie, 50.000.000 transistor e la capacità di

1. Concetti introduttivi

eseguire circa un miliardo di operazioni al secondo.

I principali parametri per la valutazione dello sviluppo e delle prestazioni sono:

- Throughput (ampiezza di banda) = corrisponde alla quantità di operazioni svolte in un'unità di tempo e si calcola

$$\frac{\text{Operazioni, Azioni, Compiti}}{\text{Tempo}}$$

- Latenza, ovvero il tempo impiegato per la completa esecuzione di un'operazione.

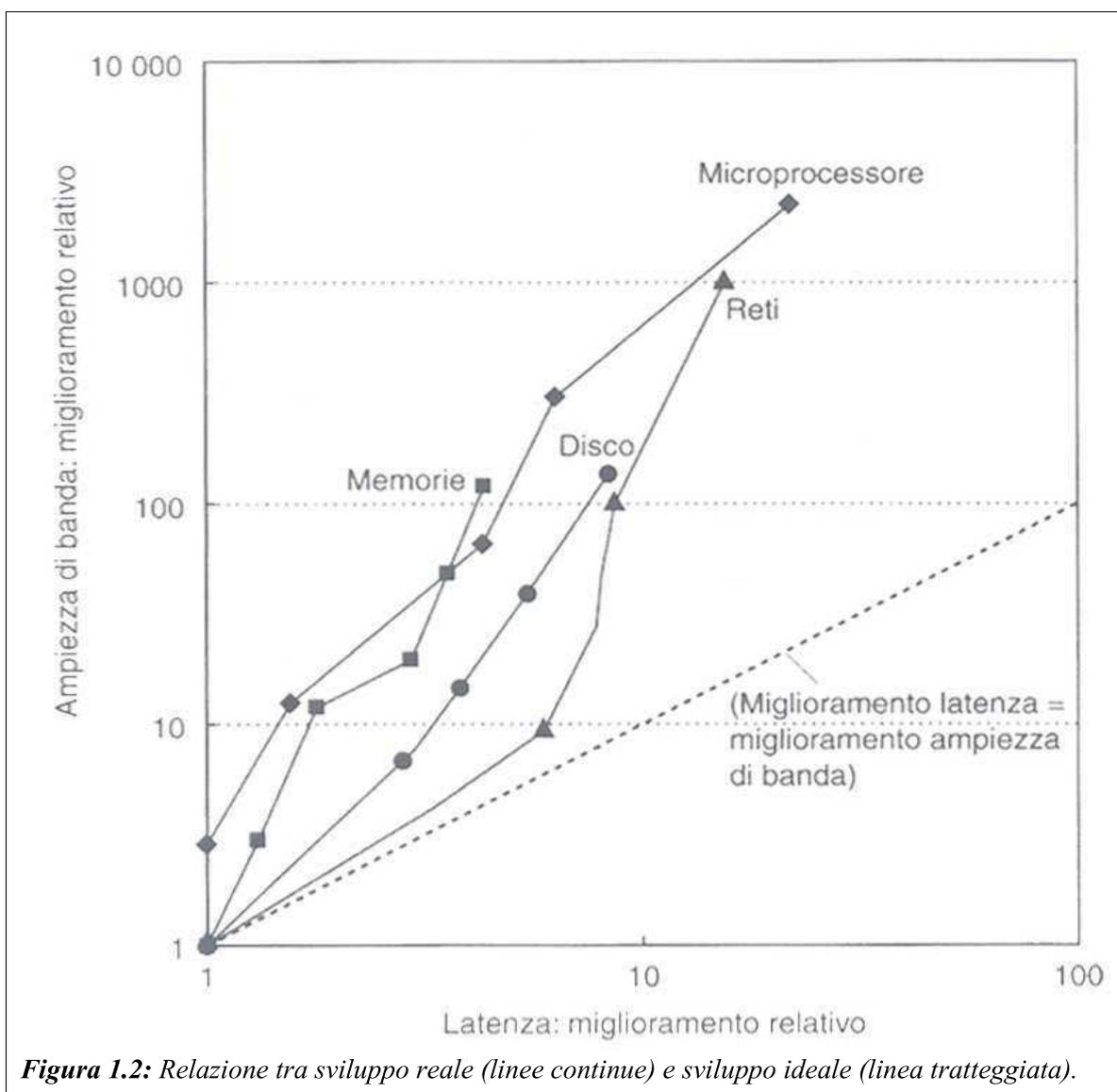


Figura 1.2: Relazione tra sviluppo reale (linee continue) e sviluppo ideale (linea tratteggiata).

Dal grafico è evidente come gli sforzi fatti in ambito evolutivo siano principalmente legati all'aumento del throughput discostandosi dalla linea ideale, dove miglioramento della latenza e del throughput hanno una proporzionalità diretta.

Nella valutazione dello sviluppo nel campo della microelettronica è molto importante la potenza dissipata.

Essa è definita come:

$$P_{Dissipata} = \frac{Energia}{Tempo} = \frac{1}{2} \cdot C_{Carico} \cdot Tensione^2 \cdot frequenza$$

Esempio:

Alcuni microprocessori vengono progettati per funzionare con tensione di alimentazione regolabile, in modo tale che una diminuzione del 15% della tensione riduce anche parimenti la frequenza del clock.

$$Riduzione\ Potenza = \frac{(Tensione \cdot 0.85)^2 \cdot Frequenza \cdot 0.85}{Tensione^2 \cdot Frequenza} = 0.61 = 61\%$$

A fronte di una riduzione delle prestazioni del solo 15% si ottiene una riduzione in potenza di circa del 60%.

Se consideriamo che nel mondo sono presenti circa 300 milioni di PC attivi con un consumo medio di 400 Kwh, avremo un consumo energetico totale di 120 Twh. Con solamente un risparmio del 10% nei PC, si risparmierebbero 12 Twh, ossia alcune centrali elettriche.

A parità di microelettronica sono stati apportati miglioramenti anche grazie all'utilizzo del parallelismo (numero di pipeline per gestire le istruzioni in parallelo) e lo sfruttamento del principio di localizzazione spaziale (inserimento nella cache delle istruzioni adiacenti in memoria a quella in esecuzione) e temporale (inserimento nella cache delle istruzioni chiamate più frequentemente).

1. Concetti introduttivi

Un errore comune è quello di considerare la CPU, in quanto a prestazioni, l'elemento più influente dell'intera architettura di un elaboratore. In realtà le prestazioni di un sistema sono determinate anche e soprattutto da :

- Architettura del sistema (BUS, gerarchia delle memorie, I/O, interrupt)
- Prestazioni e tipologia delle periferiche (rete, memorie di massa)
- Software di base

Per poter valutare i possibili miglioramenti in un sistema informatico mediante l'ottimizzazione di taluni componenti, possiamo rifarci alla legge matematica di Amdahl:

$$Speedup = \frac{Performance\ con\ il\ miglioramento}{Performance\ senza\ miglioramento}$$

Tale risultato dipende sostanzialmente da due fattori

- $Fraction_{enhanced}$: la percentuale di sistema a cui viene applicato un determinato miglioramento
- $Speedup_{enhanced}$: la grandezza del miglioramento applicato

Per comprendere come si ottiene il miglioramento complessivo del sistema, bisogna valutare il tempo di esecuzione in riferimento al vecchio tempo di esecuzione

$$ExecutionTime_{new} = ExecutionTime_{old} \cdot \left((1 - Fraction_{Enhanced}) + \frac{Fraction_{Enhanced}}{Speedup_{Enhanced}} \right)$$

Il miglioramento complessivo è dato dal rapporto del vecchio tempo rispetto al nuovo tempo di esecuzione

$$Seedup_{overall} = \frac{ExecutionTime_{old}}{ExecutionTime_{new}} = \frac{1}{(1 - Fraction_{Enhanced}) + \frac{Fraction_{Enhanced}}{Speedup_{Enhanced}}}$$

Esempi:

1. Si ha un elaboratore dove sono eseguiti dei miglioramenti tali per cui i 40% dei programmi vanno 10 volte più veloci. Qual'è il miglioramento complessivo del sistema?

$$Speedup_{overall} = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = 1.56$$

Questo dato indica come miglioramenti, anche elevati, applicati però ad una parte limitata del sistema sono assolutamente marginali. Si ha infatti un miglioramento totale sul sistema di 1.56 a fronte di un'ottimizzazione che permetteva di avere una velocità 10 volte superiore.

2. Abbiamo un elaboratore che esegue tante operazioni floating-point. Inoltre sappiamo che una delle operazioni più onerose è l'operazione di radice quadrata

- 2.1 Si focalizza l'attenzione sulla radice quadrata, riprogettando l'hardware in modo tale che l'operazione abbia una performance 10 volte superiore. In relazione al tempo dedicato al floating-point supponiamo che il tempo dedicato alla radice quadrata sia il 20% del tempo.

$$Speedup_1 = \frac{1}{(1 - 0.2) + \frac{0.2}{10}} = 1.22$$

- 2.2 Si ottimizzano del doppio (e quindi di un fattore 2) le prestazioni del floating-point, senza preoccuparsi della radice quadrata, sapendo che il 50% sono istruzioni di floating-point.

$$Speedup_2 = \frac{1}{(1 - 0.5) + \frac{0.5}{2}} = 1.33$$

Anche in questo caso è possibile notare come nel caso 2, solo un raddoppio delle prestazioni (associato però a metà del sistema) sia più incisivo rispetto al caso 1 in cui si è scelto di migliorare le prestazioni di un fattore 10 (rispetto ad 1/5 del sistema).

3. Si supponga di aumentare di 3 volte le prestazioni della memoria sapendo che le operazioni che fanno accesso alla memoria sono circa il 40%.

$$Speedup_3 = \frac{1}{(1 - 0.4) + \frac{0.4}{3}} = 1.36$$

4. In un sistema per applicazioni web il 60% del tempo è occupato dalle operazioni di I/O, mentre solo il restante 40% dall'esecuzione delle istruzioni. Sostituendo il processore di questo sistema con uno 10 volte più veloce, lo Speedup sarà pari a:

$$Speedup_4 = \frac{1}{(1 - 0.4) + \frac{0.4}{10}} = 1.56$$

Questo fa riflettere sul fatto che la potenza del processore non sia l'elemento con maggiore importanza. Per questo motivo nei progetti di server il costo del processore non è mai dominante, mentre è molto rilevante quello dei dischi.

1.3 Architettura base dei microprocessori

Un microprocessore è un implementazione fisica di un processore (CPU) realizzata su un singolo circuito integrato.

Esso è utilizzato, oltre che in quasi tutti i computer, in molti dispositivi digitali quali ad esempio telefoni, cellulari e stampanti.

Esistono tre grandi gruppi di sistemi di microprocessori:

- **General Purpose:** sono processori ad alte prestazioni e versatili, non limitati ad un solo possibile utilizzo. Fanno parte di questo gruppo i processori dei personal computer.
- **Special Purpose:** sono processori realizzati per compiti specifici. Ne fanno parte i microprocessori delle schede grafiche.
- **Embedded:** sono sistemi a microprocessore progettati per una determinata applicazione, spesso con piattaforma hardware ad hoc. Sono quindi integrati completamente nel sistema, di cui ne garantiscono tutte o quasi le funzionalità.

Ad oggi il mercato è costituito per il 98% da microprocessori per applicazioni embedded come elettrodomestici, periferiche, computer ed altro. I microprocessori per personal computer, pur rappresentando solo lo 0.2% del numero totale di pezzi prodotti, assorbono la metà del fatturato totale.

In generale, un microprocessore può essere schematizzato come segue



Figura 1.3: Schema generico di un microprocessore.

dove:

- unità di controllo: basata su microcodice o basata sui circuiti. I microprocessori di tipo RISC sono realizzati interamente da circuiti, mentre i microprocessori CISC sono composti da un 50% hardware e 50% microcodice;
- unità di decodifica istruzioni: possibilità di decodifica di più istruzioni in parallelo per andare verso un'architettura super scalare e gestione dei tempi di Fetch, Decode ed Execute;
- unità di gestione degli indirizzi: permette di realizzare in hardware il passaggio dagli indirizzi logici a quelli fisici;
- unità di gestione BUS: ha la funzione di sincronizzazione con il mondo esterno tramite i protocolli del bus stesso;
- unità esecutiva: composta da una o più ALU e interconnessa ai diversi registri.

Un altro concetto che occorre tenere presente è come avviene temporalmente l'esecuzione di un'istruzione. Essa avviene attraverso tre fasi; una fase di reperimento dell'istruzione (in molti sistemi si tende ad avere istruzioni della stessa lunghezza per poter ottimizzare i tempi di trasferimento al fine di ottenere tempi di fetch simili), una fase di decodifica e una fase di esecuzione.

L'altro elemento di riflessione è il rapporto tra il mondo esterno e quello interno. Oramai i microprocessori hanno di fatto due clock: uno verso il mondo esterno e uno verso quello interno.

Per quanto riguarda il "mondo interno" abbiamo delle frequenze più elevate, mentre nel mondo esterno abbiamo frequenze più basse che sono determinate sostanzialmente da due fattori:

- il dispositivo che è appeso è intrinsecamente più lento quindi sarebbe inutile avere frequenze elevate se il dispositivo opera su frequenze inferiori; inoltre più le frequenze sono elevate e maggiori sono i costi dei circuiti;
- quanto più le frequenze sono elevate tanto più i problemi circuitali diventano complessi; la trasmissione dei dati non avviene più per flusso di elettroni ma per onde radio.

1.4 Tecniche per incrementare le prestazioni

Per migliorare le prestazioni si sono introdotti dei miglioramenti architetturali: si è scelto di disaccoppiare il bus di sistema da quello della CPU demandando la fase di prelevamento delle istruzioni alla BIU (Bus Interface Unit) ed inserendo una coda di prefetch, che consiste nel precaricare l'istruzione successiva del programma in esecuzione; è anche possibile utilizzare una memoria cache L1 dedicata per il codice.

Si è inoltre scelto di aumentare il parallelismo di esecuzione introducendo una o più pipeline o utilizzando un'architettura super scalare, cioè un'architettura che permette di eseguire più istruzioni contemporaneamente.

Per quanto riguarda i sistemi multi-core il miglioramento delle prestazioni è conseguito grazie ad un più ampio utilizzo del parallelismo tra thread (strategia Thread Level Parallelism) piuttosto che un parallelismo tra istruzioni (Instruction Level Parallelism): quest'ultima strategia, infatti, ha bisogno di più complessità hardware oltre che di nuovi criteri di programmazione.

1. Concetti introduttivi

Di seguito sono riportati, in termini quantitativi, i miglioramenti prestazionali introdotti da alcune evoluzioni tecnologiche ed architetturali sui microprocessori. In particolare si valuteranno i seguenti elementi:

- Velocità del bus esterno
- parallelismo (prefetch, ...)
- cache L1

Per valutare le prestazioni si farà riferimento alla gestione della memoria video poiché è uno degli elementi critici nell'architettura del sistema sia in profondità nella memoria sia nell'aggiornamento della memoria video.

Caso di studio:

$$\text{Memoria Video Richiesta} = 600 \text{ pixel} \cdot 400 \text{ pixel} \cdot 3 \text{ Byte} = 0.72 \text{ MB}$$

$$\text{Refresh} = 50 \text{ Hz}$$

$$\text{Transfer Rate richiesto} = 0.72 \text{ MB} \cdot 50 \text{ Hz} = 36 \text{ MB/s}$$

$$\text{Tempo aggiornamento richiesto} = 20 \text{ ms}$$

Come tempo di esecuzione si considera il tempo necessario a fare una scrittura in memoria

1. Caratteristiche del sistema:

- Sistema sequenziale, Monobus
- Microprocessore M24 (8086/88 $f_p=8\text{MHz}$)
- Bus pre-ISA

$$\text{Ampiezza}_{\text{BUS}} = 16 \text{ bit}$$

$$f_{\text{BUS}} = 8\text{MHz}$$

- $T_{ck}=1/f_p = 125 \text{ ns}$
- $T_{\text{Accesso Memoria}}= 4 \cdot T_{ck}=500 \text{ ns}$

Elaborazione:

Calcolo del Tempo medio di istruzione, $T_{\text{Medio Istruzione}}$

Lunghezza media istruzioni₈₀₈₆=2.5 Byte

$$\left(\frac{2.5 \text{ Byte}}{\text{Ampiezza}_{BUS}} = 2 \text{ accessi in memoria per la fase di fetch} \right)$$

$$T_{\text{Medio Istruzione}}=T_{\text{Fetch}}+T_{\text{Execution}}=2 \cdot 500 \text{ ns} + 500 \text{ ns} = 1.5 \mu\text{s}$$

$$\text{Transfer Rate} = \frac{1}{T_{\text{Accesso Memoria}}} \cdot \text{Ampiezza}_{BUS} = 4 \text{ MB/s}$$

molto minore rispetto al

Transfer Rate richiesto dal problema pari a 36 MB/s

$$\text{Tempo}_{\text{Aggiornamento Medio}} = \frac{T_{\text{Medio Istruzione}}}{\text{Ampiezza}_{BUS}} \cdot \text{Memoria}_{\text{Video Richiesta}} =$$

$$\frac{1.5 \mu\text{s}}{2 \text{ Byte}} \cdot 0.72 \text{ MB} = 540 \text{ ms} \gg 20 \text{ ms}$$

Il sistema non risulta essere sufficiente

1. Concetti introduttivi

2. Caratteristiche del sistema:

- Sistema parallelo, Monobus
- Prefetch Buffer
- Faster BUS ($2 \cdot T_{ck} \rightarrow$ sfrutto fronte di salita e discesa del clock)
- Microprocessore M24 (8086/88 $f_p=8\text{MHz}$)
- Bus pre-ISA

$$\text{Ampiezza}_{BUS}=16 \text{ bit}$$

$$f_{BUS}=8\text{MHz}$$

- $T_{ck}=1/f_p = 125 \text{ ns}$
- $T_{\text{Accesso Memoria}}= 2 \cdot T_{ck}=250 \text{ ns}$

Elaborazione:

Calcolo del Tempo medio di istruzione, $T_{\text{Medio Istruzione}}$

Lunghezza media istruzioni₈₀₈₆=2.5 Byte

$$\left(\frac{2.5 \text{ Byte}}{\text{Ampiezza}_{BUS}} = 2 \text{ accessi in memoria per la fase di fetch } \right)$$

$T_{\text{Medio Istruzione}}=T_{\text{Fetch}}+T_{\text{Execution}}=\cancel{2 \cdot 500} \text{ ns} + 250 \text{ ns} = 250 \text{ ns}$ (poiché il sistema è parallelo abbiamo che a regime il tempo di esecuzione si sovrappone al tempo di fetch)

$$\text{Transfer Rate} = \frac{1}{T_{\text{Accesso Memoria}}} \cdot \text{Ampiezza}_{\text{BUS}} = 8 \text{ MB/s}$$
 ancora minore rispetto al
 Transfer Rate richiesto dal problema pari a 36 MB/s

$$\text{Tempo}_{\text{Aggiornamento Medio}} = \frac{T_{\text{Medio Istruzione}}}{\text{Ampiezza}_{\text{BUS}}} \cdot \text{Memoria}_{\text{Video Richiesta}} =$$

$$\frac{250 \text{ ns}}{2 \text{ Byte}} \cdot 0.72 \text{ MB} = 90 \text{ ms} > 20 \text{ ms}$$

Il sistema non risulta ancora essere sufficiente.

3. Caratteristiche del sistema:

- Sistema parallelo, Monobus
- Prefetch Buffer
- Faster BUS ($2 \cdot T_{\text{ck}} \rightarrow$ sfruttamento fronte di salita e discesa del clock)
- Microprocessore Pentium I ($f_p = 30 \text{ MHz}$)
- Bus pre-ISA

$$\text{Ampiezza}_{\text{BUS}} = 32 \text{ bit}$$

$$f_{\text{BUS}} = 30 \text{ MHz}$$

- $T_{\text{ck}} = 1/f_p = 33 \text{ ns}$
- $T_{\text{Accesso Memoria}} = 2 \cdot T_{\text{ck}} = 66 \text{ ns}$

Elaborazione:

Calcolo del Tempo medio di istruzione, $T_{\text{Medio Istruzione}}$

Lunghezza media istruzioni=2.5 Byte

$$\left(\frac{2.5 \text{ Byte}}{\text{Ampiezza}_{BUS}} = 1 \text{ accesso in memoria per la fase di fetch} \right)$$

$$T_{\text{Medio Istruzione}} = T_{\text{Fetch}} + T_{\text{Execution}} = 0 \text{ (prefetching)} + 66 \text{ ns} = 66 \text{ ns}$$

$$\text{Transfer Rate} = \frac{1}{T_{\text{Accesso Memoria}}} \cdot \text{Ampiezza}_{BUS} = 60 \text{ MB/s} \text{ superiore rispetto al}$$

Transfer Rate richiesto dal problema pari a 36 MB/s

Tempo Aggiornamento Medio =

$$\frac{T_{\text{Medio Istruzione}}}{\text{Ampiezza}_{BUS}} \cdot \text{Memoria}_{\text{Video Richiesta}} = \frac{66 \text{ ns}}{4 \text{ Byte}} \cdot 0.72 \text{ MB} = 11.88 \text{ ms} < 20 \text{ ms}$$

Il sistema risulta essere sufficiente.

4. Caratteristiche del sistema:

- Sistema parallelo, Multi level bus, cache
- Prefetch Buffer
- Faster BUS ($2 \cdot T_{ck} \rightarrow$ sfrutto fronte di salita e discesa del clock)
- Microprocessore ($f_p=200\text{MHz}$)
- Multi Level Bus (AGP, PCI, ISA)
- $T_{ck}=1/f_p = 5 \text{ ns}$

Elaborazione:

Il tempo di istruzione è una media tra i tempi di CPU, memoria e I/O.

Calcolo del Tempo medio di istruzione, $T_{\text{Medio Istruzione}} = \text{Media} \{T_{\text{CPU}}, T_{\text{I/O}}, T_{\text{Memoria}}\}$

- T_{CPU} è il tempo dovuto a istruzioni che impiegano solamente la CPU
- $T_{\text{I/O}}$ è il tempo dovuto a istruzioni che coinvolgono dispositivi di I/O
- T_{memoria} è il tempo dovuto a istruzioni che effettuano l'accesso in memoria

Ipotesi:

Il calcolatore esegue istruzioni ripartite nel seguente modo

- 25% tempo dedicato alle istruzioni della CPU
- 70% tempo dedicato alle istruzioni per l'accesso in memoria
- 5% tempo dedicato alle istruzioni per i dispositivi di I/O

Calcolo dei singoli tempi:

- $T_{\text{CPU}} = 5 \text{ ns}$
- T_{memoria} :

Tempo Cache (Hit/Miss=90%) = 10 ns

Tempo DRAM = 60 ns

Ipotesi 1

$T_{\text{I/O}}$:

Tempo ISA (10 Mhz, $2 \cdot T_{\text{ck}}$) = 200 ns

Si ottiene

$$\begin{aligned}\text{Tempo totale} &= 25\% \cdot 5\text{ns} + 70\% \cdot (90\% \cdot 10\text{ns} + 10\% \cdot 60\text{ns}) + 5\% \cdot 200\text{ns} \\ &= \mathbf{21.75 \text{ ns}}\end{aligned}$$

Ipotesi 2

T_{IO} :

$$\text{Tempo PCI (66 Mhz, } 2 \cdot T_{ck}) = 33 \text{ ns}$$

Otteniamo

$$\begin{aligned}\text{Tempo totale} &= 25\% \cdot 5\text{ns} + 70\% \cdot (90\% \cdot 10\text{ns} + 10\% \cdot 60\text{ns}) + 5\% \cdot 33\text{ns} \\ &= \mathbf{13.4 \text{ ns}}\end{aligned}$$

Si può notare come anche un aumento della frequenza di lavoro della CPU influisce in modo non superiore a pochi punti percentuali sul tempo totale.

Ad esempio un miglioramento alla sola CPU del 100% (T_{CPU} si di mezza) riduce solo di circa il 5% il Tempo totale.

2. Architettura x86-32

2.1 Architettura Pentium 80x86, x64

L'evoluzione delle architetture da 16 a 32 e a 64 bit ha permesso una maggiore capacità di indirizzamento della memoria. Con 16 bit, infatti, si possono indirizzare solamente 2^{16} byte (ossia 64 KB), mentre con 32 bit si ottiene una capacità di 4GB sino a raggiungere i 18EB ($1.8 \cdot 10^{19}$ byte) con architetture a 64 bit.

Il chip logico della piattaforma x86 è sostanzialmente compatibile con tre tipi di macchine: macchine a 16, 32 e 64 bit. La ragione di questa coesistenza deriva dalla necessità di compatibilità con il passato, con particolare rilievo all'architettura delle istruzioni. Per questo motivo, quando furono introdotti i primi 386 (famiglia x86-32) si doveva ricercare la compatibilità con l'8086 o il 286 (famiglia x86-16); analogamente, quando furono introdotti i primi Pentium 4 (famiglia x86-64) si doveva ricercare la compatibilità con i 386 o 486.

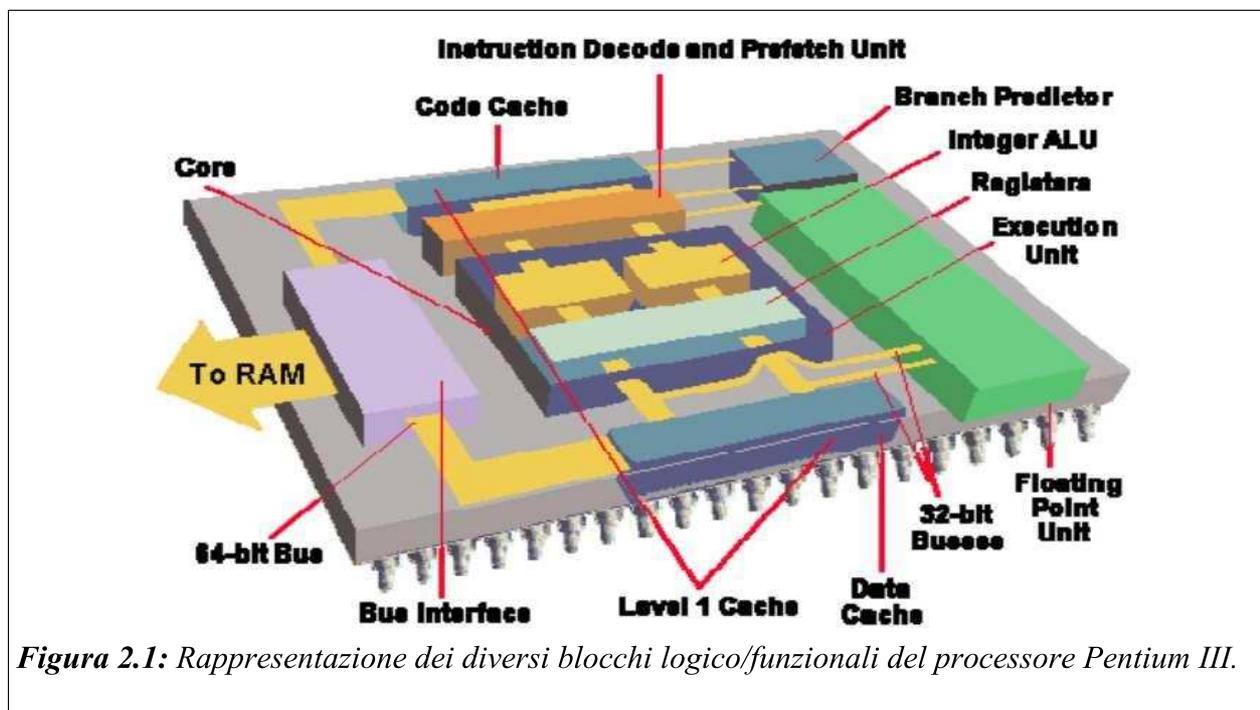


Figura 2.1: Rappresentazione dei diversi blocchi logico/funzionali del processore Pentium III.

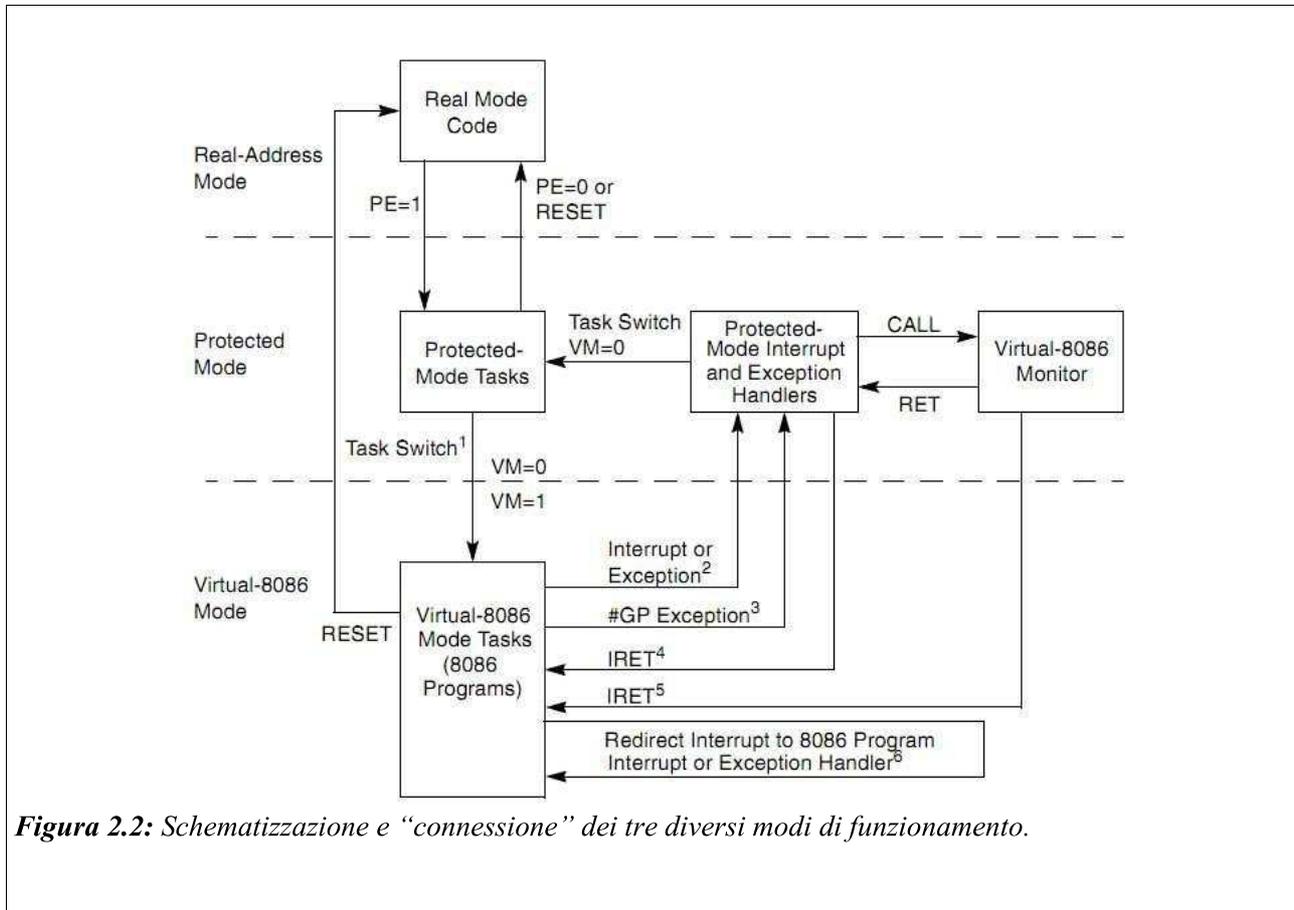
Questa figura mette in luce alcuni aspetti fondamentali in termini di occupazione di transistor per unità funzionali. È possibile notare come l'unità di floating-point abbia una dimensione maggiore di quella dei blocchi cache e quindi un numero superiore di transistor. L'altro elemento, a volte trascurato, è quello legato alla predizione dei branch il quale permette in tempo reale, durante l'esecuzione di un'istruzione, di gestire le risposte dei salti condizionati (richiede una quantità hardware non trascurabile).

2.2 Registri e modi

Tutte le architetture x86 possono operare, sostanzialmente, in tre modi diversi tra loro non compatibili:

- **Modo reale:** è la modalità in cui può essere realizzata la piena compatibilità hardware con i vecchi 8086; la gestione della memoria è piuttosto limitata in quanto si ha a disposizione poco più di 1Mbyte. Tale memoria è quella attiva al reset del processore e viene assunta dal BIOS all'avvio. Una volta caricato e lanciato il kernel avviene il passaggio nel modo protetto.
- **Modo protetto:** è il modo nativo del pentium in cui tutte le funzionalità architetturali sono compatibili con i sistemi operativi di ultima generazione (windows NT, 2000, xp, vista, Linux).
- **Modo virtuale:** la VM86 (Virtual Mode) emula in modo protetto il comportamento logico del DOS. In realtà tale emulazione avviene in due modi:
 - Modo reale eseguendo solo il set di istruzioni a 16 bit;

- In modo protetto mediante l'attivazione di un processo specifico (VM86), in cui, quando lanciato il task il sistema opera emulando il comportamento logico dell'ambiente 8086/DS.



1. Task Switch effettuato in due modi:
 - CALL o JMP dove il flag VM (Virtual Machine) nell'EFLAFGS è settato ad 1;
 - IRET dove VM e NT (nested task) sono settati ad 1.
2. Interrupt hardware o eccezione: interrupt software (INT n) quando IOPL (I/O privilege level) è pari a 3;

3. Eccezione di protezione generale causato da interrupt software (INT n) quando IOPL è minore di 3;
4. Ritorno normale di interrupt in modo protetto o gestore delle eccezioni;
5. Ritorno dal monitor 8086 per reindirizzare un interrupt o eccezione al programma 8086 in esecuzione in modalità virtuale;
6. Reindirizzamento interno di un interrupt software (INT n) quando VME è settato ad 1, IOPL è minore di 3 ed il bit di reindirizzamento è settato ad 1.

L'architettura di un pentium o di una singola CPU di un *core* è sostanzialmente costituita da un insieme di registri. Tali registri possono essere da 8, 16, 32 bit; essi sono:

- **Data Register**

utilizzati per memorizzare operandi e risultati delle operazioni

Registro	Descrizione	Utilizzato anche	Note per l'Assembler
EAX/ AX/ AH AL	Accumulator Register		Moltiplicazione, Divisione, I/O, Shift veloce
EBX/ BX/ BH BL	Base Register	Per calcolo indirizzi	
ECX/ CX/ CH CL	Count Register	Come contatore	Cicli, Rotazioni, Shift
EDX/ DX/ DH DL	Data Register	Per operazioni di I/O come puntatore	Moltiplicazione, Divisione, I/O

- **Pointer Register**

utilizzati come puntatori

Registro	Descrizione	Funzione	Note per l'Assembler
EIP / IP	Instruction Pointer	Punta alla successiva istruzione da eseguire	Non modificabile dall'utente
ESI / SI	Source Index	Registri indice per la memoria	Da trattare come operandi di tipo <i>mem</i>
EDI / DI	Destination Index	Registri indice per la memoria	Da trattare come operandi di tipo <i>mem</i>

- **Stack Register**

utilizzati per l'implementazione dello stack

Registro	Descrizione	Funzione	Note per l'Assembler
ESP / SP	Stack Pointer	Punta alla testa dello stack	L'indirizzo viene decrementato ad ogni PUSH e incrementato ad ogni POP
EBP / BP	Base Pointer	Base address dello stack	

- **Test Register**

utilizzati nella fase di Self-Test

Registro	Descrizione
TR7	Test Data
TR6	Test command

- **Segment Register**

il modello architetturale di fondo della memoria è segmentato

Registro	Registri Offset validi	Descrizione	Funzione	Note per l'Assembler
DS, ES	EBX / ESI / EDI / BX / SI / DI	Data Segment	Indirizzo base del segmento dati	
CS	EIP / IP	Code Segment	Indirizzo base del segmento codice	Per modificare tale registro è necessaria una chiamata a procedura <i>far</i> , oppure una <i>far jump</i> oppure un interrupt. In Protected Mode viene verificato se il nuovo segmento può essere utilizzato dal task.
SS	ESP / EBP / SP / BP	Stack Segment	Indirizzo del segmento stack	
FS, GS		Extra Segment	Segmento extra per i dati, ad uso generico	

- **Flags Register**

L' x86 dispone di un registro dei flag, chiamato EFLAG. Sebbene non sia direttamente accessibile mediante un nome (come EAX, ad esempio) è possibile in qualche modo leggerlo e scriverlo. Esempio:

```
/* Lettura dello stato del registro dei flags */
PUSHFD          ;Salva lo stato dei flags nello stack
```

```

POP EAX      ;Estrae dallo stack e salva in EAX (o qualunque altro
              registro generale)

/* Modifica dello stato dei flags */

PUSH EAX     ;Salva il registro EAX sullo stack (o qualunque altro
              registro generale)

POPFD       ;Estrae dallo stack e memorizza nel registro dei flags

```

- **Debug Register**

Registro	Descrizione
DR0 / DR1 / DR2 / DR3 / DR4 / DR5 / DR6 / DR7	Debug Register

- **Control Register**

Registro	Descrizione
CR0 / CR1 / CR2 / CR3	Control Register

- **Memory Management Registers**

Registro	Descrizione
GDTR	Global Descriptor Table Register
IDTR	Interrupt Descriptor Table Register
TR	Task Register
LDTR	Local Descriptor Table Register

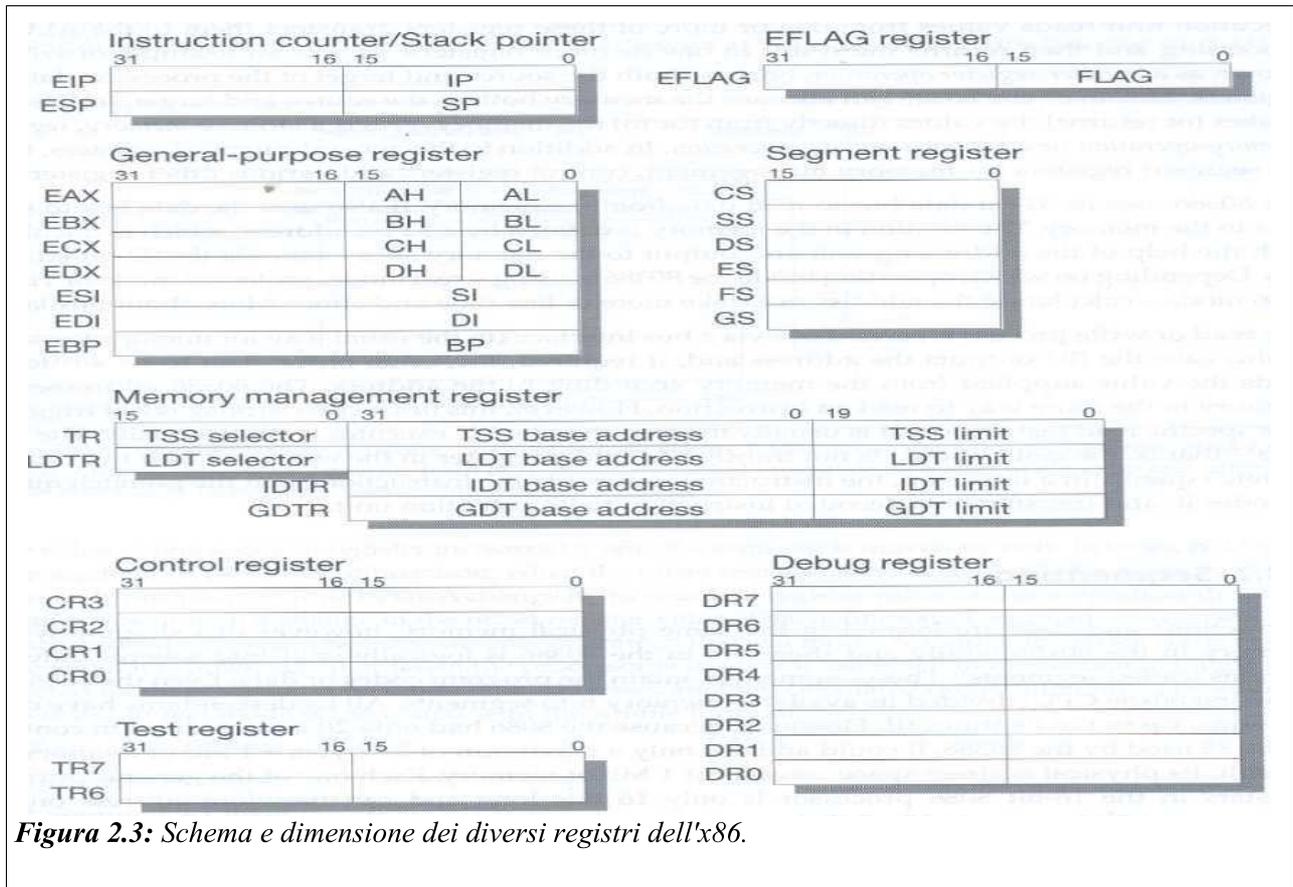


Figura 2.3: Schema e dimensione dei diversi registri dell'x86.

A seconda della modalità in cui il sistema opera vi è il coinvolgimento o meno di alcuni registri. In modo reale diventano attivi solamente i registri di debug, di controllo (escluso il CR0) ed i registri di gestione della memoria. Quando la macchina viene avviata, essa è sostanzialmente una macchina a 16 bit, senza memoria virtuale, senza memoria logica ma con la memoria segmentata.

VM86			Protected Mode			Real Mode		
Registro	Descrizione	Bit	Registro	Descrizione	Bit	Registro	Descrizione	Bit
IP	Instruction Pointer	16	EIP	Instruction Pointer	32	EIP	Instruction Pointer	32
SI	Source Index	16	ESI	Source Index	32	ESI	Source Index	32
DI	Destination Index	16	EDI	Destination Index	32	EDI	Destination Index	32
FLAG	Status Flags	16	EFLAG	Status Flags	32	EFLAG	Status Flags	32
AX	Accumulator	16	EAX	Accumulator	32	EAX	Accumulator	32
BX	Base	16	EBX	Base	32	EBX	Base	32
CX	Counter	16	ECX	Counter	32	ECX	Counter	32
DX	Data	16	EDX	Data	32	EDX	Data	32
SP	Stack Pointer	16	ESP	Stack Pointer	32	ESP	Stack Pointer	32
BP	Base Pointer	16	EBP	Base Pointer	32	EBP	Base Pointer	32
CS	Code Segment	16	ECS	Code Segment	16	ECS	Code Segment	16
SS	Stack Segment	16	SS	Stack Segment	16	SS	Stack Segment	16
DS	Data Segment	16	DS	Data Segment	16	DS	Data Segment	16
ES	Extra Segment	16	ES	Extra Segment	16	ES	Extra Segment	16
			FS	Extra Segment	16	FS	Extra Segment	16
			GS	Extra Segment	16	GS	Extra Segment	16
						TR	Task	16 32 16
						LDTR	Local Desc. Table	16 32 16
						IDTR	Interrupt Desc. Table	32 16
						GDTR	Global Desc. Table	32 16
						CR3	Control	32
						CR2	Control	32
						CR1	Control	32
						CR0	Control	32
			TR7	Test	32	TR7	Test	32
			TR6	Test	32	TR6	Test	32
						DR7	Debug	32
						DR6	Debug	32
						DR5	Debug	32
						DR4	Debug	32
						DR3	Debug	32
						DR2	Debug	32
						DR1	Debug	32
						DR0	Debug	32

Figura 2.4: Tabella riassuntiva dei registri coinvolti nei diversi modi.

I registri dati, nella nostra architettura di riferimento, possono essere da 8, 16 oppure 32 bit, e vengono denominati con dei criteri fissi. Viene specificato come prima lettera il nome del registro (per esempio A → Accumulator), seguito da:

Nel caso di un'opzione su un byte da H (es. AH per indicare la parte più significativa del registro “high”) oppure da L (es. AL per indicare la parte meno significativa del registro “low”);

Nel caso di un'opzione su 16 bit e quindi 2 byte il nome del registro viene seguito dalla

lettera X (es. AX).

In altri casi particolari, dove per esempio i registri sono su 32 bit, allora si antepone la lettera E (es. EAX “extended”). Nel caso invece dei registri a 64 bit, essi assumono un nome diverso che è R.

I registri di controllo e di indirizzo contengono essenzialmente degli indirizzi esprimibili su 16 o 32 bit. Come già visto, un parallelismo su 16 bit implica la possibilità di accesso ad una memoria di dimensione 64Kbyte, mentre un parallelismo dei registri a 32 permette di gestire memorie dell'ordine dei 4Gbyte.

La dimensione del address-bus è fortemente legato alla dimensione dei registri. Esso, infatti, deve avere un numero di bit non inferiore al massimo numero di bit di un registro. In modo implicito possiamo affermare che in presenza di un bus a 32 bit, non potrà avere registri con un parallelismo superiore.

Esempio e considerazioni:

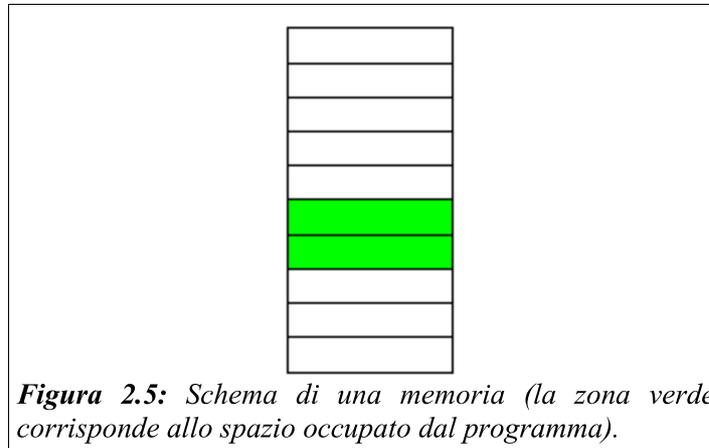
Per comprendere meglio la gestione e la dimensione delle varie istruzioni nella memoria e i diversi problemi ad essi legati, vediamo un esempio sul problema del debugging.

Tipicamente un debugger è un ambiente software che permette una serie di opzioni di cui due fondamentali: la prima è la possibilità di inserire dei *breakpoint* (il che significa che impostando un'istruzione o un indirizzo al programma, quando uno di questi viene raggiunto si ritorna al debugger), la seconda è la possibilità di eseguire un debug *step-by-step* (ad ogni istruzione eseguita vi è il ritorno al debugger).

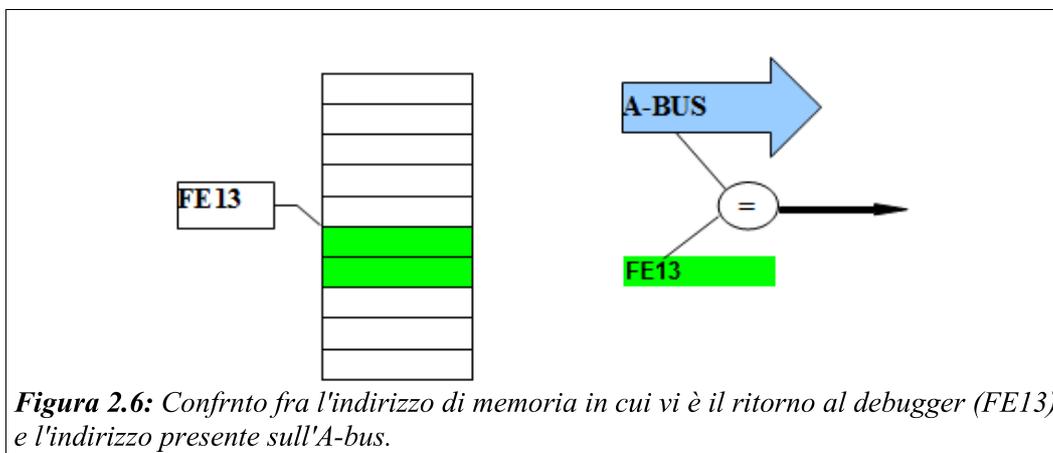
Supponendo di operare in modo protetto, analizziamo come realizzare queste funzioni e i problemi ad essi annessi.

- Breakpoint:

Ipotizziamo che la zona verde della memoria in cui è memorizzato il nostro programma rappresenti il breakpoint.



Una possibile soluzione, per effettuare il “salto” al debugger, potrebbe essere quella di confrontare l'indirizzo a cui è memorizzata l'istruzione di ritorno al programma (FE13 nel nostro esempio) e l'indirizzo presente sull'*adress-bus*.



Se, mediante un confronto, il valore degli indirizzi risulta essere uguale, allora blocco il sistema e ritorno nell'ambiente di debugging. Per realizzare questa metodologia è necessario introdurre dell'hardware apposito che prelevi i segnali dall'address-bus per poter eseguire il confronto tra indirizzi. Questa strategia, risulta essere troppo complicata e macchinosa.

Una possibile alternativa consiste nel sostituire l'istruzione del programma con una jump o una call al debugger in corrispondenza del punto in cui vogliamo inserire il breakpoint.

Questo implica che sarà necessario salvare le istruzioni che si andranno a sostituire.

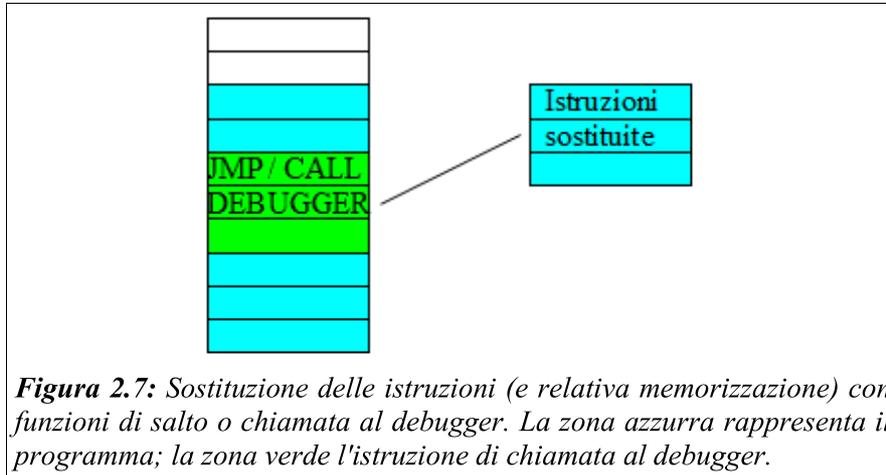
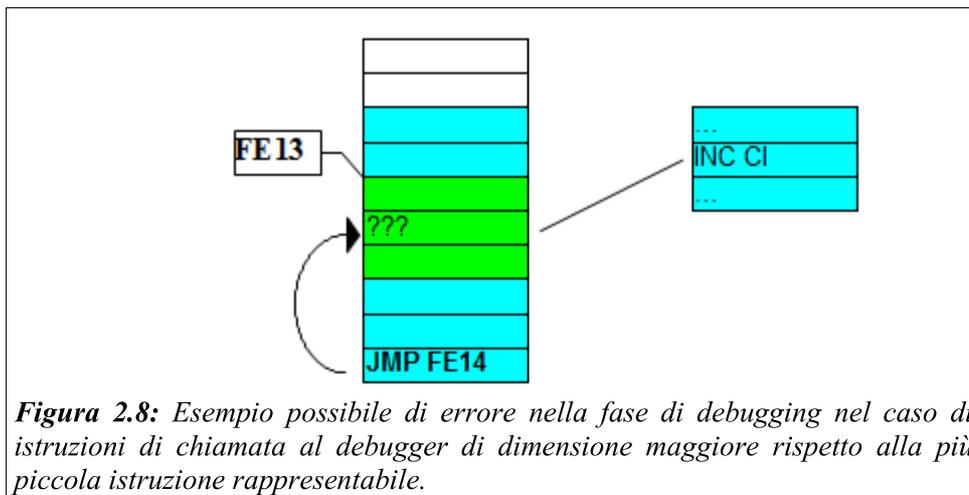


Figura 2.7: Sostituzione delle istruzioni (e relativa memorizzazione) con funzioni di salto o chiamata al debugger. La zona azzurra rappresenta il programma; la zona verde l'istruzione di chiamata al debugger.

A questo punto, eseguendo il programma, in corrispondenza della “vecchia istruzione” e quindi nel punto da noi scelto per eseguire il breakpoint, vi sarà un salto al debugger.

Non bisogna però trascurare la dimensione della procedura di salto. Infatti, ipotizzando di utilizzare una memoria di 64Kbyte, si hanno indirizzi su 16 bit; questo comporta, nel nostro esempio, che l'istruzione di *jump* sarà definita almeno su 3 byte. In termini pratici questo implica una sostituzione nella memoria di 3byte di istruzioni in corrispondenza del punto in cui voglio eseguire il breakpoint. Questo può comportare dei problemi; infatti se per esempio in fondo al codice si ha un'istruzione di salto nella zona che è stata sostituita (zona verde), il programma andrà ad operare su istruzioni diverse da quelle che si aspettava.



Nasce quindi l'obbligo di utilizzare un'istruzione di *jump* “particolare” che deve essere espressa su un numero di byte pari al minimo numero di byte su cui può essere espressa un'istruzione. Nella piattaforma x86 il numero minimo di byte per un'istruzione è 1byte (esempio NOP → *not operation*).

L'unico modo per poter effettuare un salto su un byte è quello di utilizzare un indirizzo implicito. È necessario che a livello architetturale sia introdotto un posto nella memoria, noto dal sistema, nel quale sarà inserito l'indirizzo a cui si dovrà eseguire il salto (si esegue un salto indiretto). Nel caso dell'architettura x86, questa istruzione prende il nome di INT3, la quale indica di saltare ad un indirizzo fisso che corrisponde alla quarta posizione nel vettore delle interruzioni.

- Step-by-step:

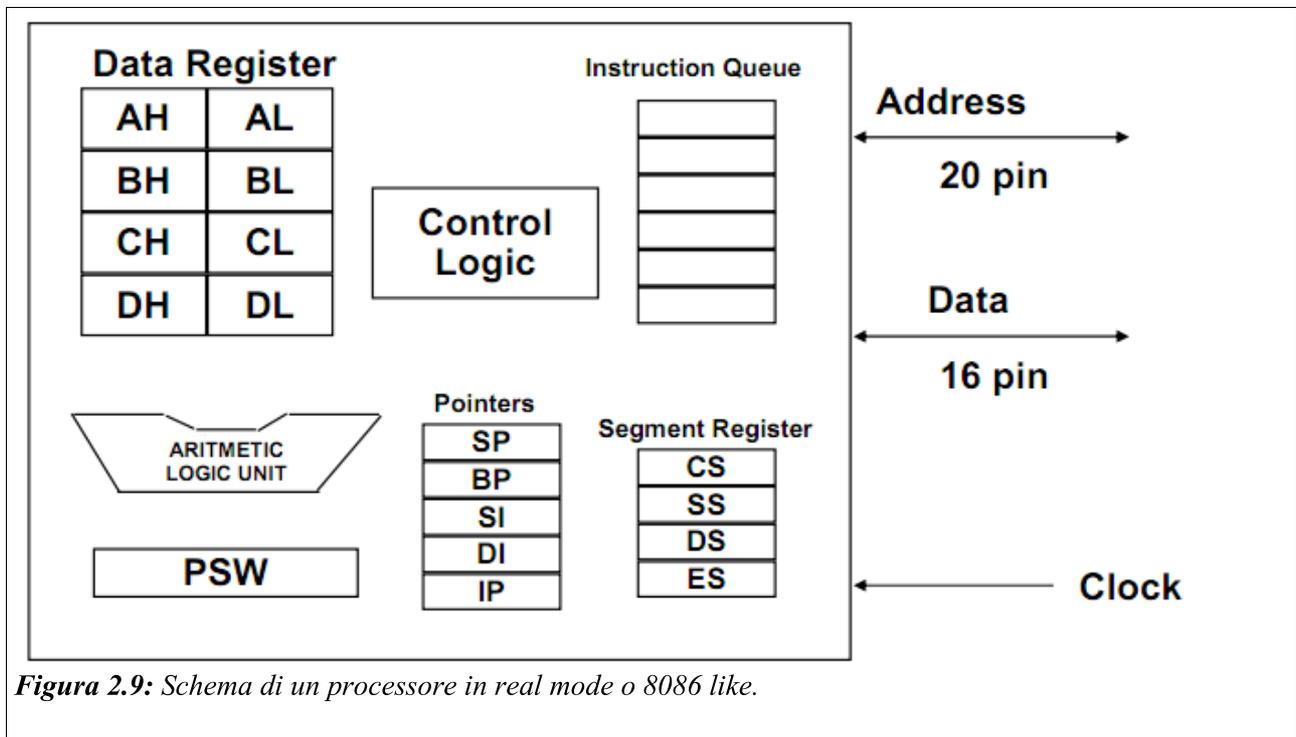
Eeguire il debug step-by-step significa ritornare al debugger dopo l'esecuzione di ogni istruzione.

Una possibile soluzione potrebbe essere quella di adottare la strategia del breakpoint. Nota la posizione dell'istruzione i -esima, inserire un breakpoint nell'istruzione $i+1$ -esima e così via.

Nell'architettura x86 si ha la possibilità di settare un bit nel registro di controllo chiamato *trap bit*. Tale bit, automaticamente al termine di una istruzione, fa scattare un interrupt. Al termine dell'istruzione in corso d'esecuzione si va a servire la procedura di interrupt la cui routine di servizio corrisponde al mio debugger.

2.3 Gestione della memoria nel modo reale (segmentazione)

Il modo reale, chiamato anche *8086 like*, è stato progettato per duplicare l'ambiente di esecuzione del processore intel 8086. In questa modalità si ha un'architettura logica così definita:



Tutta l'architettura opera su un parallelismo pari a 16bit, in quanto tutti gli operandi nel “codice nativo 8086” sono espressi su valori a 8 o 16bit. L'architettura è caratterizzata da quattro registri di segmento (tipicamente sono 6, ma il modo reale ne vede solamente 4): CS, DS, SS ed ES dove CS contiene l'indirizzo del segmento di codice, DS ed ES gli indirizzi dei segmenti di dato ed SS l'indirizzo del segmento di stack. Sono presenti cinque registri puntatore: SP, BP, SI, DI, IP tutti espressi su 16 bit dove IP (instruction pointer) contiene il puntatore alla prima istruzione da eseguire, SP (stack pointer) contiene il puntatore alla testa dello stack, BP (base pointer) viene utilizzato come base per fare l'accesso all'interno dello stack ed SI (Source Index) e DI (Destination Index) vengono utilizzati come registri indice. Sono presenti i registri di dato A, B, C, D nelle loro versioni a 8 e 16bit.

È inoltre presente un registro di controllo chiamato PSW (process status word / CR0). Esso è composto da 16 bit, ma solo nove di questi sono utilizzati. Ogni bit corrisponde ad un flag i quali si suddividono in:

- Flag di condizione
- Flag di controllo

				OF	DF	IF	TF	SF	ZF		AF		PF		CF
--	--	--	--	----	----	----	----	----	----	--	----	--	----	--	----

I flag di condizione vengono automaticamente scritti al termine di varie operazioni e sono:

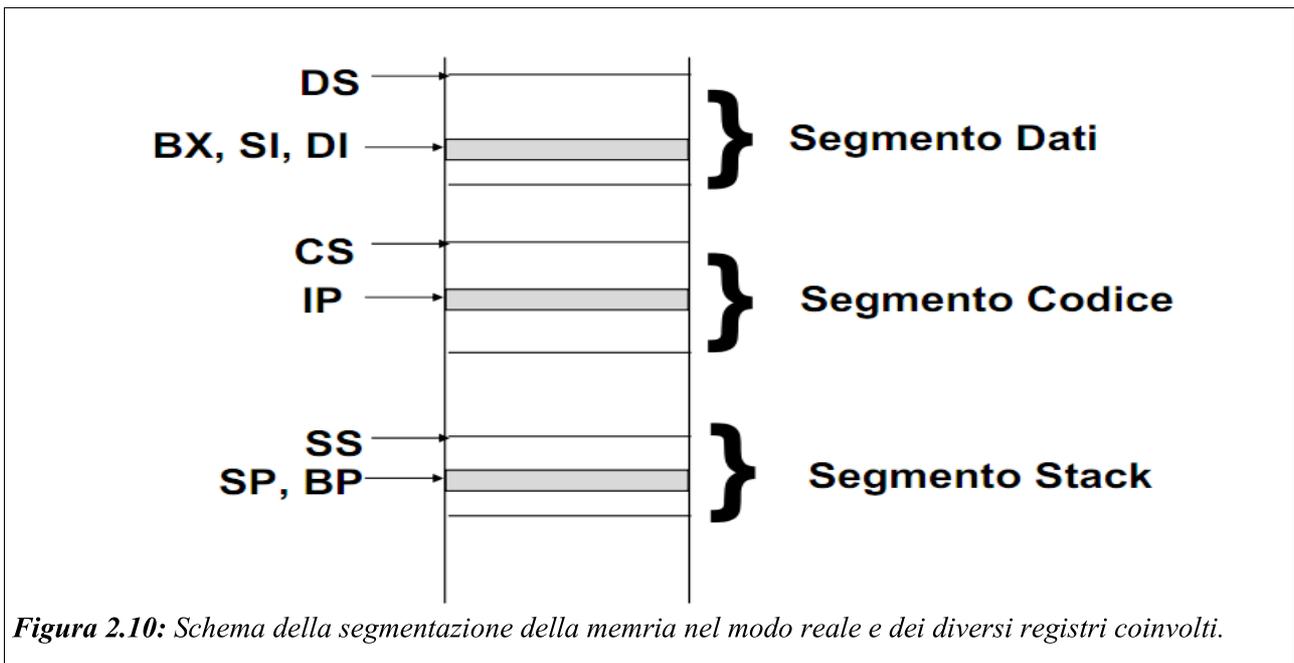
- SF (Sign Flag): coincide con il MSB del risultato dopo una operazione aritmetica;
 - ZF (Zero Flag): vale 1 se il risultato è nullo, 0 altrimenti;
 - PF (Parity Flag): vale 1 se il numero di 1 negli 8 bit meno significativi del risultato è pari , 0 altrimenti;
-

- CF (Carry Flag): dopo le operazioni aritmetiche vale 1 se c'è stato riporto (somma) o prestito (sottrazione) – altre istruzioni ne fanno un uso particolare;
- AF (Auxiliary Carry Flag): usato nell'aritmetica BCD – vale 1 se c'è stato riporto (somma) o prestito (sottrazione) – altre istruzioni ne fanno un uso particolare;
- OF (Overflow Flag): vale 1 se l'ultima istruzione ha prodotto un overflow.

I flag di controllo possono venire scritti e manipolati da apposite istruzioni, e servono a regolare il funzionamento di talune funzioni del processore:

- DF (Direction Flag): utilizzato dalle istruzioni per la manipolazione delle stringhe; se vale 0 le stringhe vengono manipolate partendo dai caratteri all'indirizzo minore, se vale 1 a partire dall'indirizzo maggiore;
- IF (Interrupt Flag): se vale 1, i segnali di interrupt mascherabili vengono percepiti dalla CPU, altrimenti questi vengono ignorati;
- TF (Trap Flag): se vale 1, viene eseguito una *trap* al termine di ogni istruzione.

Il processore (in modo *8086 like*) supporta un valore nominale di 1Mbyte di spazio di indirizzamento fisico. Tale spazio è suddiviso in segmenti, ognuno dei quali può avere una dimensione massima di 64Kbyte. La base di un segmento viene individuata mediante un selettore di segmento su 16bit seguito da quattro zeri, per un indirizzo totale su 20bit (in grado di indirizzare 1Mbyte di memoria). Poiché la dimensione massima di un segmento è pari a 64Kbyte, è possibile accedere ad un operando all'interno del segmento aggiungendo un offset di 16bit rispetto alla base del segmento. Un indirizzo fisico è perciò formato dalla base del segmento (espressa su 20bit) sommato all'offset (espresso su 16bit).



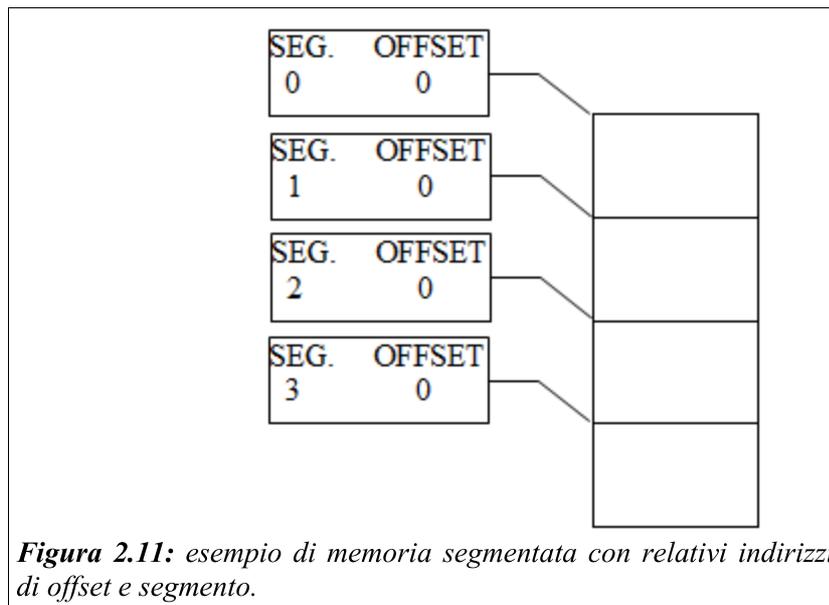
La tecnica della segmentazione insieme alla tecnica di paginazione (che nel modo reale non è attivabile) è una tecnica di gestione della memoria che è nata negli anni '60-'70 sui grandi mainframe. La segmentazione nasce da un fondamentale concetto: il programma è composto da una serie di entità che afferiscono a zone diverse della memoria. Le entità fondamentali di un programma sono il codice, i dati e lo stack. Queste entità logiche sono mappate in porzioni distinte della memoria. Al fine di poter gestire con flessibilità queste entità, si individuano nella memoria dei blocchi di memoria per l'appunto dei segmenti.

Mediante la suddivisione della memoria in segmenti si possono ottenere diversi vantaggi:

- Spazio di indirizzamento pari a 2^{20} , ma indirizzi su 16 bit;
- Separazione tra dati, codice e stack;
- Possibilità di avere più segmenti dello stesso tipo (dati, codice o stack);
- Possibilità di sovrapposizione tra segmenti, con minimizzazione della memoria sprecata;
- Rilocabilità.

Esempio e considerazioni:

Se la nostra memoria è suddivisa in blocchi, chiamati segmenti, è necessario avere due porzioni dell'indirizzo: una porzione che identifica il segmento e una porzione che identifica l'offset all'interno del segmento stesso. Un indirizzo, quindi, sarà dato dalla combinazione di queste due entità.



Per questo motivo, la prima cella sarà identificata come segmento 0 e offset 0, la prima cella del secondo segmento come segmento 1 e offset 0 e così via. A fronte di questo, dal punto di vista architetturale, sono necessari due registri: un registro di segmento e uno di offset. In generale si avranno macchine con registri di segmentato con lunghezza N bit essendo 2^N il numero di segmenti possibili e registri di offset di lunghezza M bit essendo 2^M il numero di byte che ciascun segmento dispone.

Maggiore è il numero di segmenti e più alta è la flessibilità ma, a pari memoria, maggiore è il numero di segmenti minore è la profondità di ciascun segmento.

Per questo tipo di architettura sono necessarie due tipi di CALL o JUMP:

- intra-segment: per salti o chiamate ad istruzioni che fanno riferimento allo stesso segmento;
- inter-segment: per salti o chiamate ad istruzioni che fanno riferimento a segmenti diversi. In tal caso sarà necessario specificare sia offset sia segmentato dell'istruzione richiesta.

La memoria segmentata può essere gestita fisicamente in modi diversi: segmenti sovrapposti e segmenti non sovrapposti.

- Segmenti non sovrapposti:

In questo tipo di architettura il segmento successivo segue fisicamente quello precedente; inoltre la lunghezza dei segmenti è fissa.

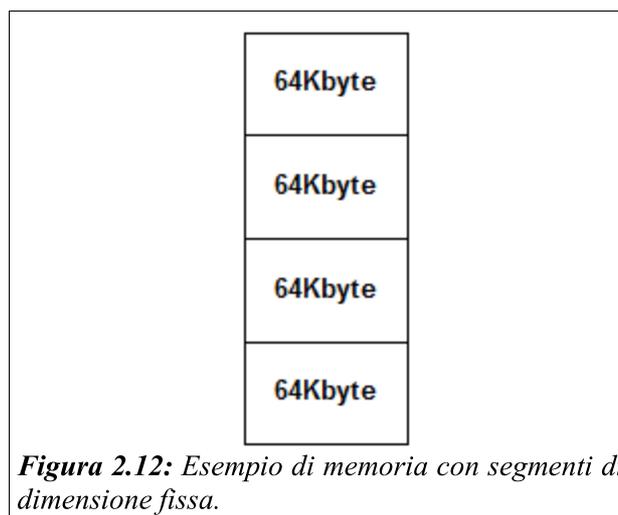
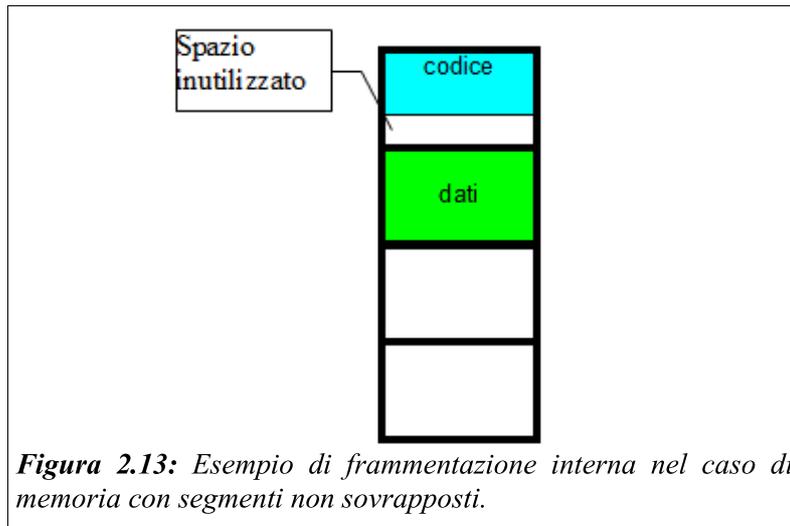
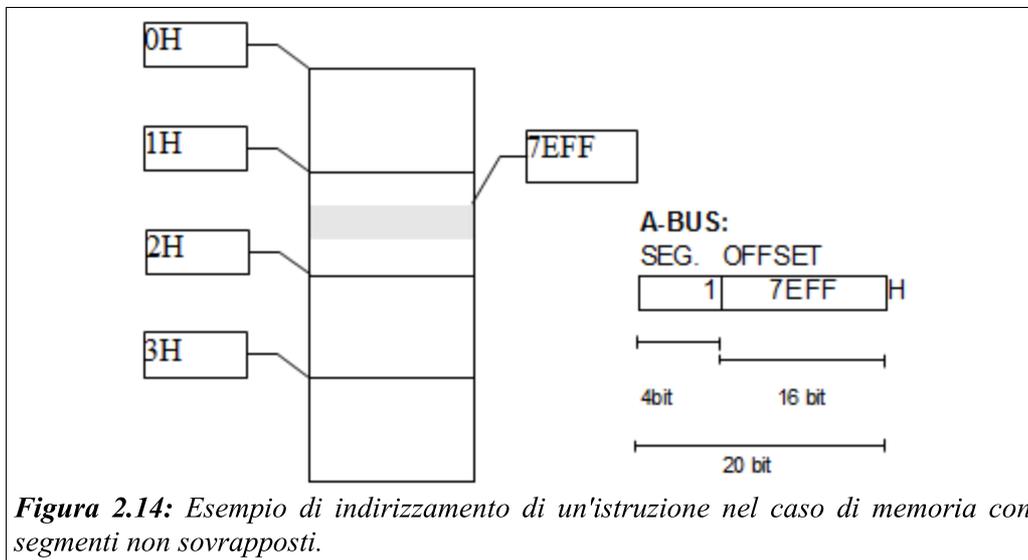


Figura 2.12: Esempio di memoria con segmenti di dimensione fissa.

Se si suppone di avere un segmento di codice di dimensione pari a 36Kbyte, si è in grado di allocare tale segmento in un unico blocco (poiché ha una dimensione pari a 64Kbyte). Se successivamente è necessario allocare un segmento dati (che non può stare nello stesso segmento di codice), l'unica possibilità è quella di caricare i dati in un nuovo segmento lasciando inutilizzato $64\text{Kbyte} - 36\text{Kbyte} = 28\text{Kbyte}$ di memoria, ossia circa un terzo dello spazio di un segmento.



Per poter indirizzare tutta la memoria seguendo questo modello, gli indirizzi su 20 bit sono organizzati in modo che i primi 4 bit identifichino il segmento e i restanti 16 bit l'offset all'interno del segmento stesso.



Il modello dei segmenti non sovrapposti è concettualmente semplice, ma non ottimizza l'occupazione della memoria fisica.

- Segmenti sovrapposti:

Il modello dei segmenti sovrapposti, adottato dal modo reale (8086 like) o in generale da tutte le architetture x86, consiste nel definire l'indirizzo di testa del segmento in una qualunque posizione della memoria.

Se si vuole realizzare un segmento di 10Byte, si andrà a definire l'indirizzo di testa del nuovo segmento dopo 10Byte.

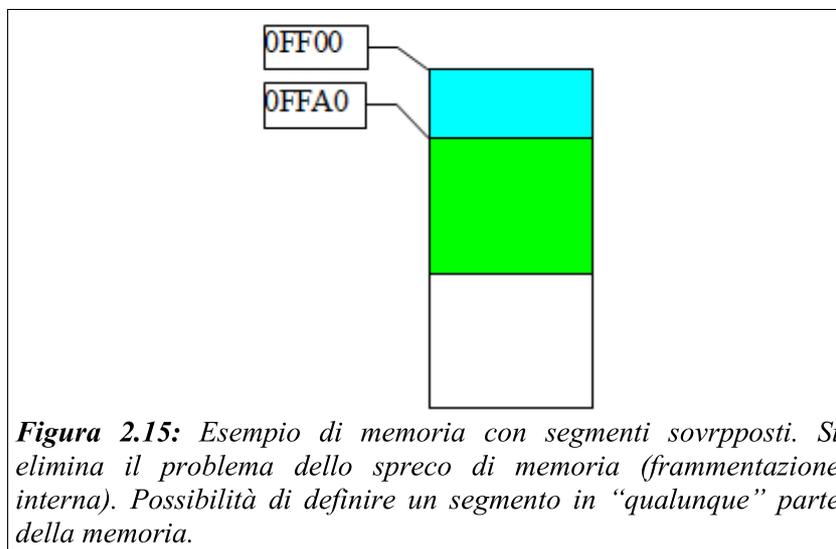
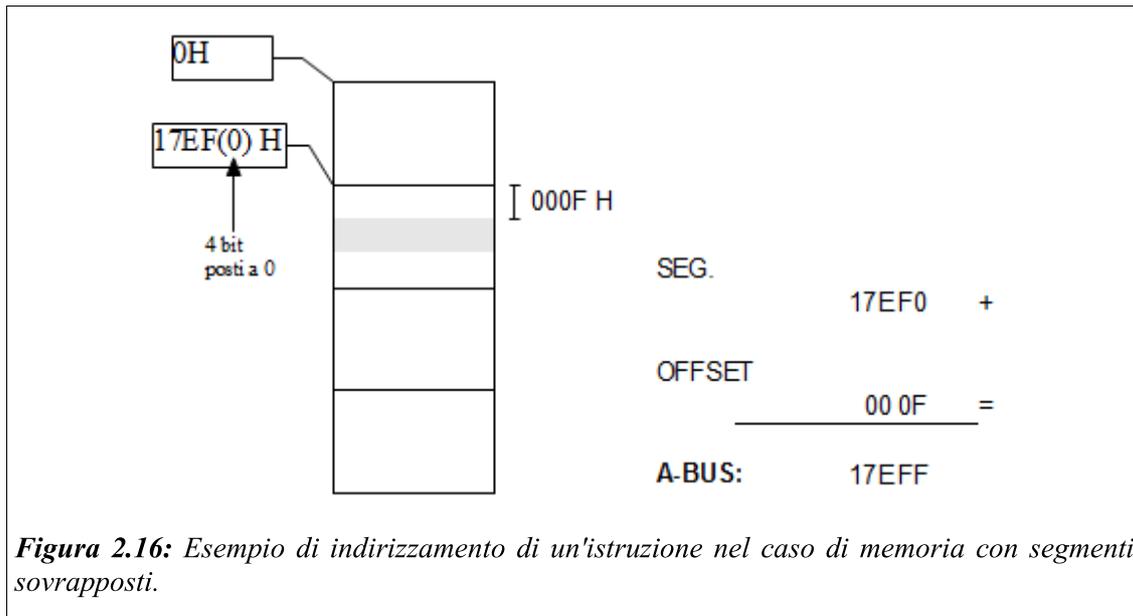


Figura 2.15: Esempio di memoria con segmenti sovrapposti. Si elimina il problema dello spreco di memoria (frammentazione interna). Possibilità di definire un segmento in “qualunque” parte della memoria.

Affinché si abbia una sovrapposizione completa, e quindi la possibilità di incominciare un segmento in qualunque punto della memoria è necessario esprimere l'indirizzo di segmento su 20 bit poiché la memoria ha una profondità di 1Mbyte. Nonostante il bus degli indirizzi sia su 20 bit, i registri hanno una dimensione massima di 16bit. Per realizzare, quindi, tale indirizzo si aggiunge in coda (ad un indirizzo di segmento di partenza espresso su 16bit) una successione di quattro 0. Questo significa che, tecnicamente, non si potrà far partire il segmento da qualunque locazione di memoria ma soltanto in locazioni sfasate di $2^4=16$ Byte

(16Byte rispetto ad 1Mbyte è trascurabile). Definito, quindi, l'indirizzo di testa del segmento su 20bit, è possibile accedere alle diverse locazioni dello stesso segmento aggiungendo un offset di 16bit (poiché la dimensione massima di questi blocchi è 64Kbyte).

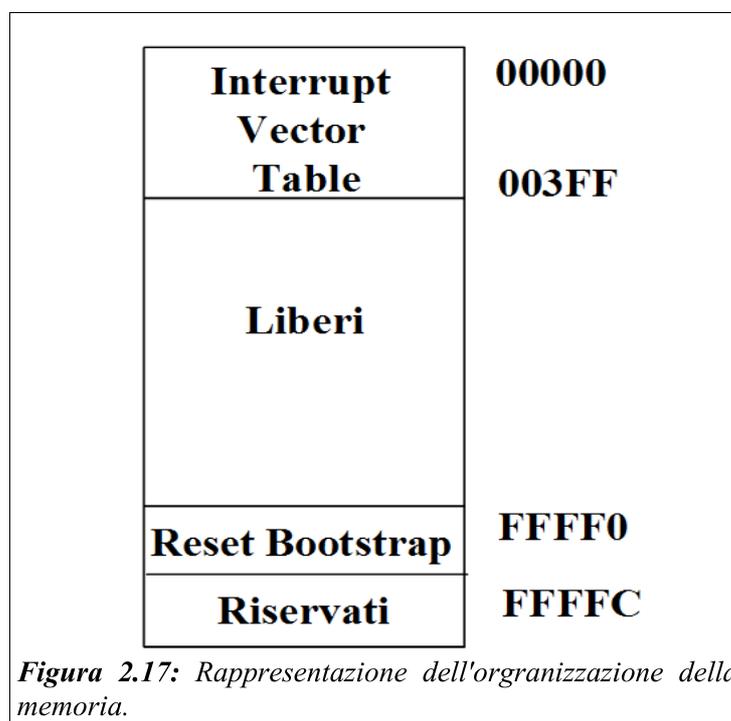


Questo fa sì che la generazione di un indirizzo sull'A-bus avvenga nel seguente modo:

- si considera uno qualunque dei registri di segmento (CS, SS, DS, ES);
- si aggiungono quattro bit posti a 0 all'indirizzo di segmento considerato;
- si considera l'indirizzo posto sul registro IP e lo si somma all'indirizzo precedentemente ottenuto <seg.>:<offset> (per es. CS:IP);
- si ottiene l'indirizzo finale espresso su 20 bit che rappresenta il program counter.

2.4 Organizzazione della memoria

La memoria è organizzata secondo il modello little endian, che fa sì che i byte meno significativi siano memorizzati negli indirizzi più bassi e i byte più significativi negli indirizzi più alti. All'interno della memoria devono esserci due spazi funzionali distinti l'entry point e la interrupt vector table.



2.4.1 Entry point

All'avvio del processore il program counter è inizializzato ad un valore arbitrario, occorre forzare tale valore con un ben preciso indirizzo. Tale indirizzo solitamente si trova nella parte bassa o nella parte alta della memoria; è fondamentale che all'indirizzo con cui viene inizializzato il program counter corrisponda una zona di memoria non volatile, tale che, anche dopo un'operazione

di reset contenga le routine necessarie per l'avvio del processore: il BIOS. Nel caso dei processori Intel l'indirizzo a cui viene forzato il program counter è coincidente con l'indirizzo più alto della memoria esclusi 16 byte, che corrisponde ad un indirizzo fisico su 20 bit: 0FFFF0h.

Quindi dopo la ricezione del segnale di reset l'8086 si procura il codice della prima istruzione da eseguire all'indirizzo (a 20 bit) 0FFFF0H, cioè 16 byte prima della fine della sua memoria. In quella locazione è presente una memoria elettronica non volatile, ROM o EPROM. La prima istruzione farà saltare ad un piccolo programma di test e di caricamento, residente in ROM. Questo programma, normalmente detto POST (Power On Self Test) verificherà il funzionamento dei sottosistemi del computer, e leggerà dall'hard disk la parte del Sistema Operativo che deve rimanere sempre in memoria, dopo averla caricata la farà eseguire.

La sequenza di accensione di un computer, e il relativo caricamento iniziale del Sistema Operativo, viene anche detta sequenza di bootstrap¹.

Nella parte alta della memoria sono, inoltre, scritti i driver dei periferici.

2.4.2 Interrupt vector table

Le procedure di interrupt vengono gestite tramite il vettore delle interruzioni, il quale contiene gli indirizzi reali della routine a cui saltare per servire la richiesta di interrupt. Nel caso dell'x86

¹Curiosità: Nel romanzo fantastico "Le avventure del Barone di Munchhausen " (R.E. Raspe , 1785) il protagonista riesce a volare semplicemente tirando forte sui lacci dei suoi stivali (bootstraps!). Il termine sequenza di "bootstrap", usato per indicare la sequenza di avviamento di un computer, fa riferimento a questo racconto. Come il Barone di Munchausen si "autosollewa" con i suoi stessi lacci, così un computer, durante il "bootstrap", carica il suo software con il suo stesso software.

contiene 256 posizioni di 4 byte ciascuna, per un totale di 1024 byte. Per realizzare questo meccanismo occorre posizionare tale vettore in una porzione ben precisa della memoria che nel modello x86 corrisponde alla parte più bassa: a partire dall'indirizzo 00000h fino all'indirizzo 003FFh.

Di seguito una tabella dettagliata relativa alla suddivisione della Interrupt Vector Table.

Vector No.	Description	Real-Address Mode	Virtual-8086 Mode	Intel 8086 Processor
0	Divide Error (#DE)	Yes	Yes	Yes
1	Debug Exception (#DB)	Yes	Yes	No
2	NMI Interrupt	Yes	Yes	Yes
3	Breakpoint (#BP)	Yes	Yes	Yes
4	Overflow (#OF)	Yes	Yes	Yes
5	BOUND Range Exceeded (#BR)	Yes	Yes	Reserved
6	Invalid Opcode (#UD)	Yes	Yes	Reserved
7	Device Not Available (#NM)	Yes	Yes	Reserved
8	Double Fault (#DF)	Yes	Yes	Reserved
9	(Intel reserved. Do not use.)	Reserved	Reserved	Reserved
10	Invalid TSS (#TS)	Reserved	Yes	Reserved
11	Segment Not Present (#NP)	Reserved	Yes	Reserved
12	Stack Fault (#SS)	Yes	Yes	Reserved
13	General Protection (#GP)*	Yes	Yes	Reserved
14	Page Fault (#PF)	Reserved	Yes	Reserved
15	(Intel reserved. Do not use.)	Reserved	Reserved	Reserved
16	Floating-Point Error (#MF)	Yes	Yes	Reserved
17	Alignment Check (#AC)	Reserved	Yes	Reserved
18	Machine Check (#MC)	Yes	Yes	Reserved
19-31	(Intel reserved. Do not use.)	Reserved	Reserved	Reserved
32-255	User Defined Interrupts	Yes	Yes	Yes

2.4.3 Riservati

La zona più alta della memoria, corrispondente alle locazioni FFFFCh e FFFFh è riservata ai dati di versione del BIOS.

2.5 Stack

Lo stack è una struttura dati che risiede nella memoria principale, si tratta di un'area di memoria nella quale l'accesso ai dati avviene in modo LIFO (Last In First Out). Una struttura LIFO è tale che l'ultimo elemento che vi è stato depositato è anche il primo che viene ripreso e cancellato. Questo rende lo stack ideale per la memorizzazione provvisoria di informazioni che è necessario mantenere momentaneamente in memoria in attesa di essere riutilizzate in seguito.

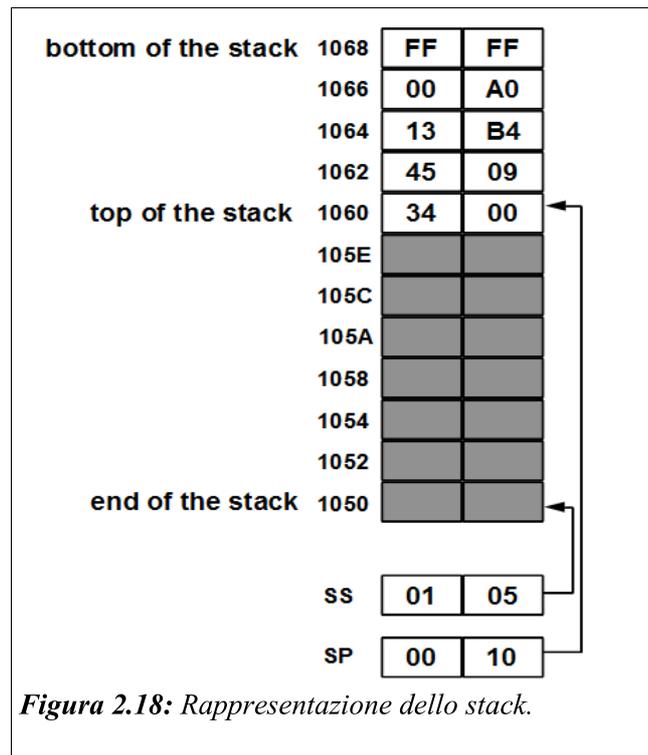
Si lavora sullo stack con due sole istruzioni: PUSH e POP.

L'istruzione di PUSH immette un nuovo valore all'interno dello stack; l'istruzione di POP lo toglie.

Entrambe le istruzioni operano trasferendo due byte per volta (16 bit).

Allo stack fanno riferimento i due registri SS ed SP.

Il registro SS (Stack Segment) punta al segmento contenente lo stack; il registro SP (Stack Pointer) punta sempre all'ultimo elemento dello stack, contiene, quindi, l'indirizzo di memoria dell'ultimo valore che vi è stato immesso.



Ogni volta che viene eseguita un'operazione di PUSH la CPU fa automaticamente, nell'ordine, le seguenti operazioni:

1. Aggiornamento dello stack pointer, che viene decrementato di 2 byte per poter effettuare la scrittura nella posizione corretta dello stack.
2. Scrittura della parola che si vuole immettere nello stack all'indirizzo indicato dallo stack pointer.

Quindi un'istruzione di tipo PUSH viene tradotta come segue:



Quando viene eseguita un'operazione di POP vengono eseguite in automatico le seguenti operazioni:

1. Lettura della locazione di memoria indicata dallo stack pointer.
2. Aggiornamento del valore dello stack pointer, il quale viene incrementato di 2 byte.

L'operazione di POP viene invece tradotta come:

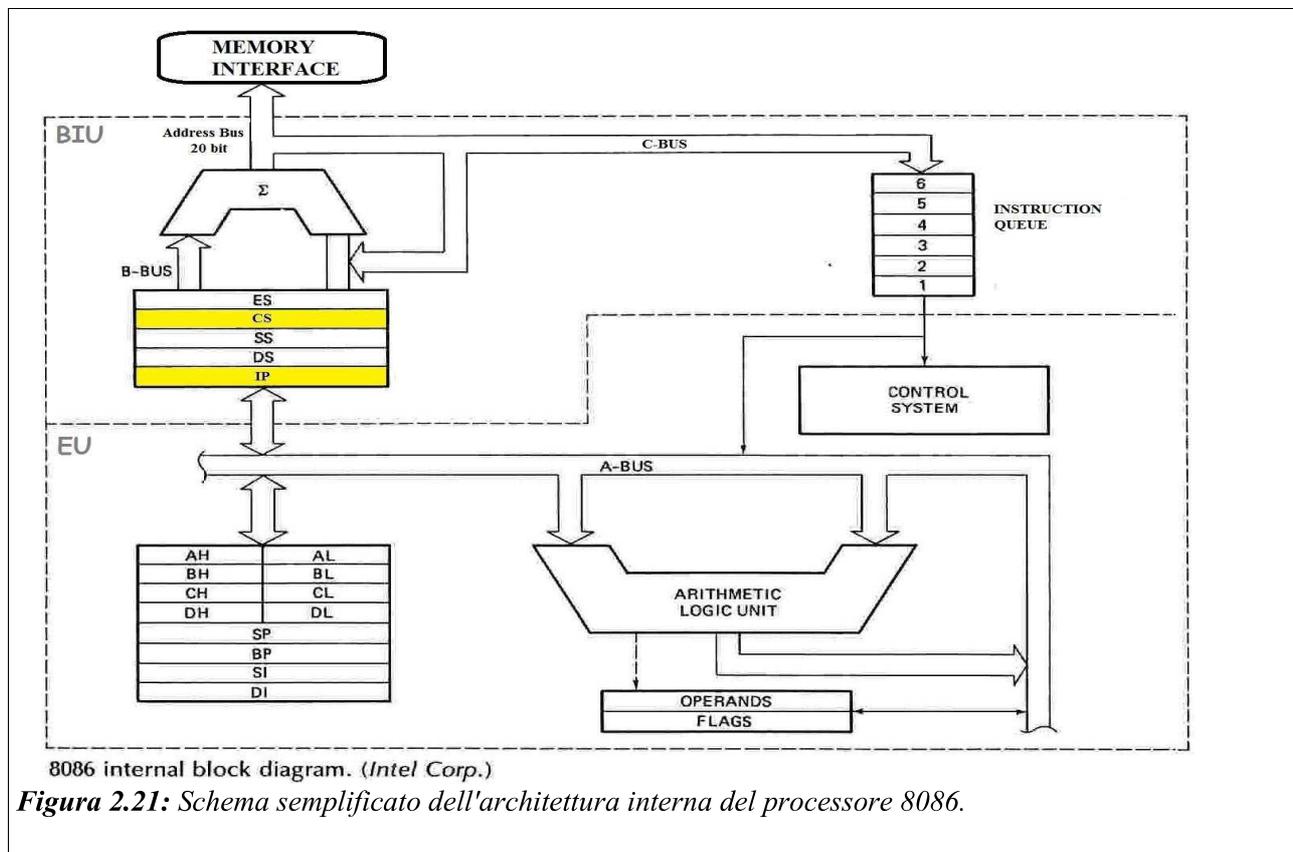


E' interessante notare che lo stack dell'8086 procede all'indietro, infatti quando viene creato SP viene inizializzato ad un valore alto, che poi, dopo ogni operazione di PUSH viene decrementato. Questo permette di rendersi conto facilmente di quando lo stack si esaurisce ("Stack overflow"). Infatti se si continuano ad eseguire operazioni di PUSH, il valore dello stack pointer cala fino ad arrivare a 0. Se ciò accade significa che lo spazio dello stack è esaurito.

A questo punto è necessario interrompere il programma. Infatti, qualora lo stack sia esaurito e si effettui un ulteriore PUSH, si esegue la scrittura in un'area di memoria al di fuori di quella riservata allo stack. Questo può provocare gravi danni, come il crash di sistema o la corruzione di dati senza che nessuno se ne accorga.

E' buona norma, soprattutto in programmi recursivi, controllare lo stack overflow confrontando il registro SP con il valore 0, dopo ogni operazione di PUSH.

2.6 Architettura interna semplificata



L'immagine 2.21 rappresenta l'architettura vera e propria del processore 8086. Per effettuare il caricamento di un'istruzione, ossia la sua fase di fetch, occorre inviare la coppia di registri CS e IP sull'interfaccia di memoria. Si noti che sia CS che IP sono espressi su 16 bit, mentre l'address bus è a 20 bit, si deve, quindi, effettuare un'operazione di scalamento e di somma; tale operazione viene eseguita da un adder dedicato (Σ) il quale ha la funzione di passare dalla coppia di registri che esprimono l'indirizzo segmentato a quello che è il valore fisico che viene inviato sull'address bus. Dopodiché viene letta dalla memoria l'istruzione che viene accumulata nella coda di prefetch (instruction queue); se la coda è vuota l'istruzione caricata sarà inviata all'unità di controllo per essere eseguita. Supponiamo che l'istruzione sia:

MOV AX, [BX].

L'unità controllo analizza l'istruzione e capisce che si tratta di una MOV, il che implica la lettura di una cella di memoria, più precisamente della cella di memoria il cui indirizzo è DS:BX.

Contemporaneamente alla fase di decodifica i bus possono rimanere inattivi. Ciò accade quando l'istruzione che è stata presa durante la fase di fetch lavora solo su registri interni. Il fatto che in questi casi i bus non vengano utilizzati si può sfruttare per rendere la CPU più veloce. Invece di lasciare i bus inattivi si può leggere il codice operativo dell'istruzione successiva a quella che viene correntemente eseguita e immagazzinarlo in una coda all'interno della CPU. Quando la fase di execute sarà terminata, la CPU potrà prendere il codice operativo della prossima istruzione dalla coda invece che dalla memoria. Considerando che la coda è già all'interno della CPU questo comporterà un grande risparmio di tempo. Ciò significa che la fase di fetch e quella di execute si possono svolgere contemporaneamente, almeno in alcuni momenti. Questo meccanismo è detto "prefetching"; la cache di istruzioni di un 8086 è detta "coda di prefetch" ed è una struttura di tipo FIFO grande 6 Byte. Dunque un 8086 durante l'esecuzione delle istruzioni che non coinvolgono la memoria legge le istruzioni successive a quella puntata dal Program Counter e le memorizza nella coda di prefetch, se essa non è già piena. Tutto ciò avviene in modo completamente automatico e trasparente al programmatore.

Per poter funzionare in questo modo l'8086 è suddiviso in due parti: una, detta Execution Unit (EU), sovrintende alla sola fase di esecuzione, mentre la Bus Interface Unit (BIU) è la parte che fa l'accesso ai bus e che, essendo indipendente dalla EU, è in grado di fare il prefetching, contemporaneamente alla fase di execute.

A questo punto, si ha la coda di prefetch parzialmente piena, mentre viene decodificata l'istruzione. Ora occorre effettuare un accesso a memoria perciò la fase di prefetch deve essere

interrotta perché occorre inviare sull'address bus l'indirizzo coerente al valore DS:BX. Quindi vengono inviati all'address bus i valori di DS e di BX con i quali verrà prodotto l'indirizzo a 20 bit nell'address bus, che verrà percepito dalla memoria, la quale risponderà inviando nel data bus il valore richiesto che deve essere salvato nel registro AX; quindi una volta che sul data bus è presente il valore recuperato dalla memoria esso verrà scritto nel registro interno AX.

Nel caso di un'istruzione che richiede di effettuare un'operazione tra registri, ad esempio:

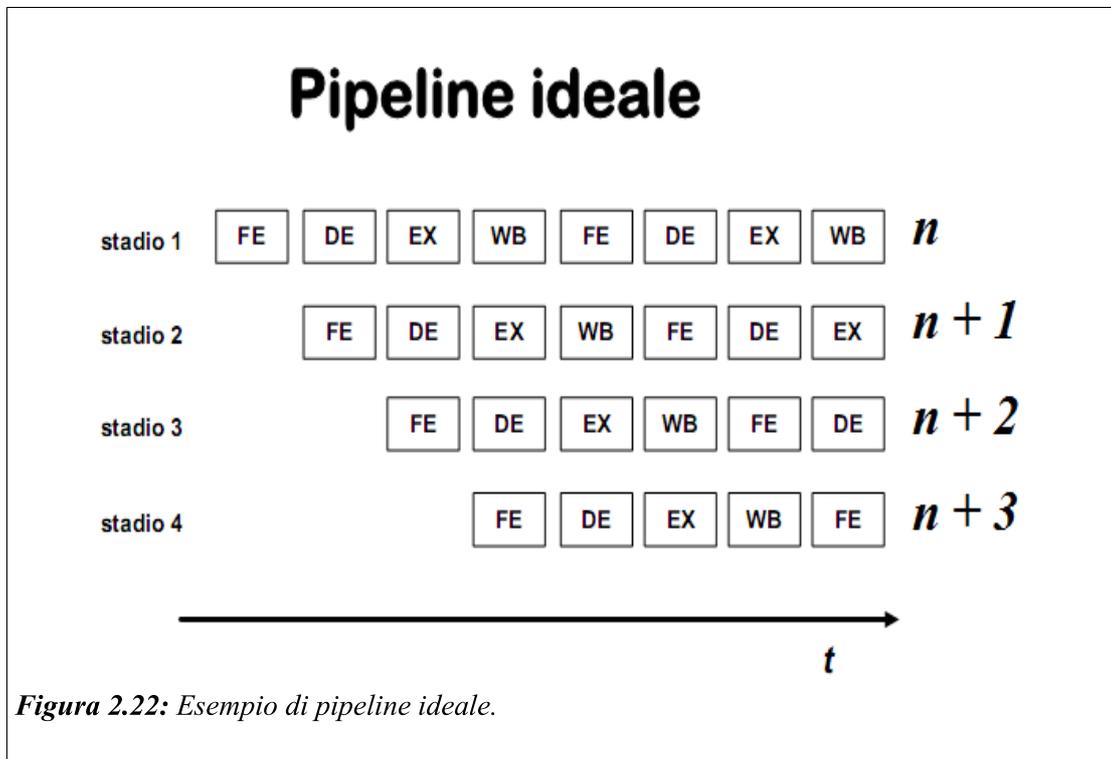
ADD AX, CX.

L'operazione di fetch risulta uguale a quella descritta precedentemente ma dopo la fase di decodifica occorre attivare la ALU per poter eseguire l'operazione di somma. Vengono quindi prelevati i valori dei registri AX e CX e inviati all'indirizzo di interfaccia dell'Arithmetic Logic Unit, sulla quale si attiva il segnale hardware corrispondente all'operazione di somma. In uscita alla ALU si ottiene il risultato; contemporaneamente all'uscita del risultato si settano i flag interessati dall'operazione di somma (ad esempio lo zero flag e il carry flag etc). Il risultato sarà poi inviato all'indirizzo di destinazione: AX (determinato dall'istruzione stessa).

Nel caso di istruzione di jump la coda di prefetch deve essere azzerata, poiché la jump invalida il contenuto di tutta la coda delle istruzioni, le quali sono ordinate sequenzialmente in funzione del valore del program counter.

2.7 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 dati, quindi l'istruzione, percorre tutti gli elementi della pipeline.



Il numero di stadi di pipeline dipende concettualmente dal numero di porzioni in cui si riesce a dividere il lavoro complessivo di un'istruzione. A seconda del modo in cui si effettua la suddivisione della singola istruzione si possono ottenere pipeline ideali o imperfette. Si suppone di aver diviso l'istruzione in quattro elementi atomici (non ulteriormente divisibili), come mostrato in figura 2.22, che corrispondono alle fasi di fetch, decodifica, esecuzione e scrittura dei risultati.

Con questa divisione si possono sovrapporre alcune attività dell'istruzione corrente con alcune dell'istruzione successiva. Non appena termina la fase di fetch dell'istruzione n viene avviata la fase di fetch dell'istruzione $n+1$, in quanto i registri e le unità funzionali utilizzate nella fase di prelevamento sono rimaste libere, e quindi possono essere utilizzate.

E' importante notare che ciascuno stadio ha, però, una durata temporale differente, ad esempio lo stadio di fetch ha un durata maggiore rispetto allo stadio di decodifica, come mostrato in figura 2.23.

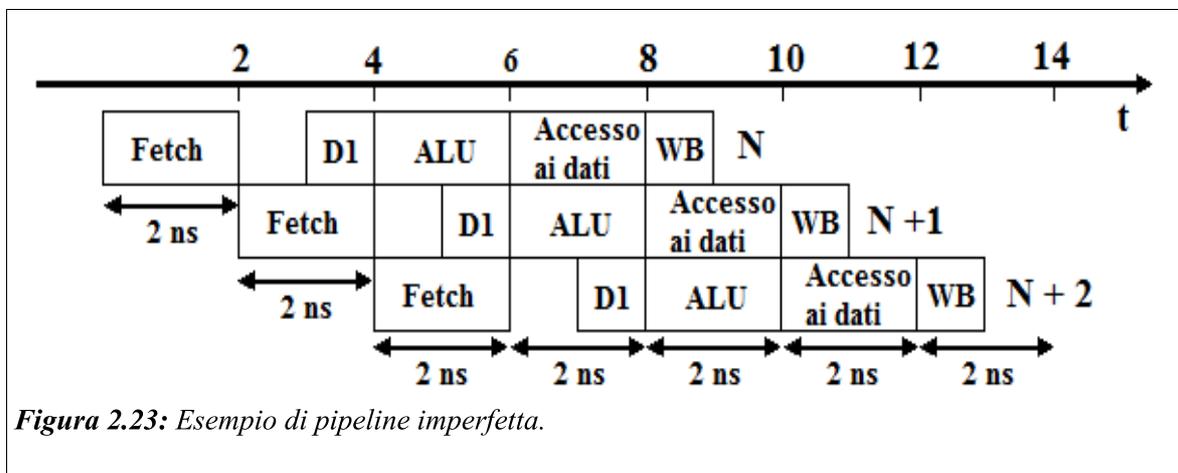


Figura 2.23: Esempio di pipeline imperfetta.

Ciò significa che il tempo minimo di durata di uno stadio è determinato dal blocco che impiega più tempo per essere completato. In figura 2.23 si nota che la fase di fetch dell'istruzione $N+1$ ha una durata maggiore rispetto alla fase di decodifica dell'istruzione N . Poiché le istruzioni sono atomiche è impossibile dividere la fase di fetch dell'istruzione $N+1$, perciò si ha una pipe imperfetta, nella quale occorre uniformare il tempo di esecuzione di ciascuno stadio. Nell'esempio in figura tutti gli stadi durano $2 ns$, nonostante alcune fasi durino $1 ns$. Questo implica che il tempo di esecuzione della singola istruzione, in caso di pipeline imperfetta, aumenta.

2.7.1 Tempo di esecuzione e throughput

Il tempo di esecuzione di un'istruzione è l'insieme di tutti tempi necessari a completare l'istruzione stessa; nell'architettura pipeline questo tempo non viene ottimizzato, ma bensì aumenta.

Il throughput corrisponde al numero di istruzioni eseguite nell'unità di tempo (MIPS), questo parametro viene fortemente ottimizzato dall'architettura pipeline.

Siano:

- 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.

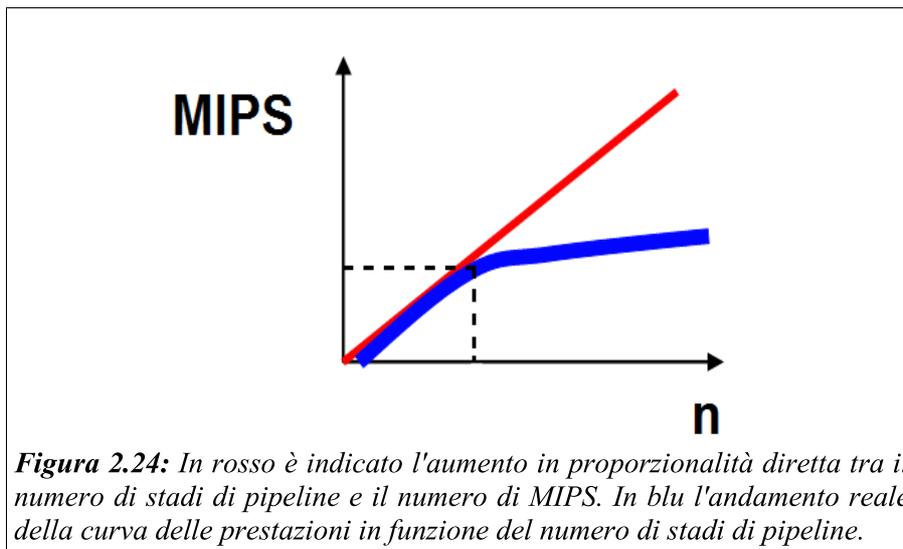
Si ha che $T_S \leq T_P$: infatti, il tempo di esecuzione di un singolo stadio è dimensionato dal tempo massimo di esecuzione nello stadio tra tutte le istruzioni del processore e pertanto la durata di una singola istruzione in una architettura pipeline può risultare maggiore.

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

- $MIPS_S = \frac{1}{T_S}$ (sistema senza pipeline)
- $MIPS_P = \frac{n}{T_P}$ (sistema con pipeline), con n numero di stadi

Si ha che $MIPS_S \ll MIPS_P$.

Si nota che il throughput, nel caso di sistema con pipeline, è proporzionale al numero di stadi di pipeline, più è alto il numero degli stadi tanto maggiore è il throughput. Ma non è possibile incrementare il numero di n a piacere poiché oltre una certa soglia tale operazione non risulta più efficiente.



2.7.2 Valutazione delle prestazioni

Siano:

- k : il numero di stadi della pipeline
- τ_i : tempo dell' i -esimo stadio
- τ_{MAX} : il più grande dei tempi τ_i

Considerando N istruzioni, si ha:

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

Dove il primo termine $k \cdot \tau_{MAX}$ indica il tempo della prima istruzione che deve essere eseguita completamente prima di portare la pipe a regime ed il secondo $(N-1) \cdot \tau_{MAX}$ indica quando sono completate le istruzioni una dopo l'altra.

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.

$$\text{Fattore di Accelerazione} = \frac{N \cdot k \cdot \tau_{MAX}}{K \cdot \tau_{MAX} + (N - 1) \cdot \tau_{MAX}}$$

Per N sufficientemente grande, quindi se si ha un alto numero di istruzioni, il fattore di accelerazione tende al valore di k. Questo dimostra che le prestazioni sono strettamente legate al numero di stadi di pipeline.

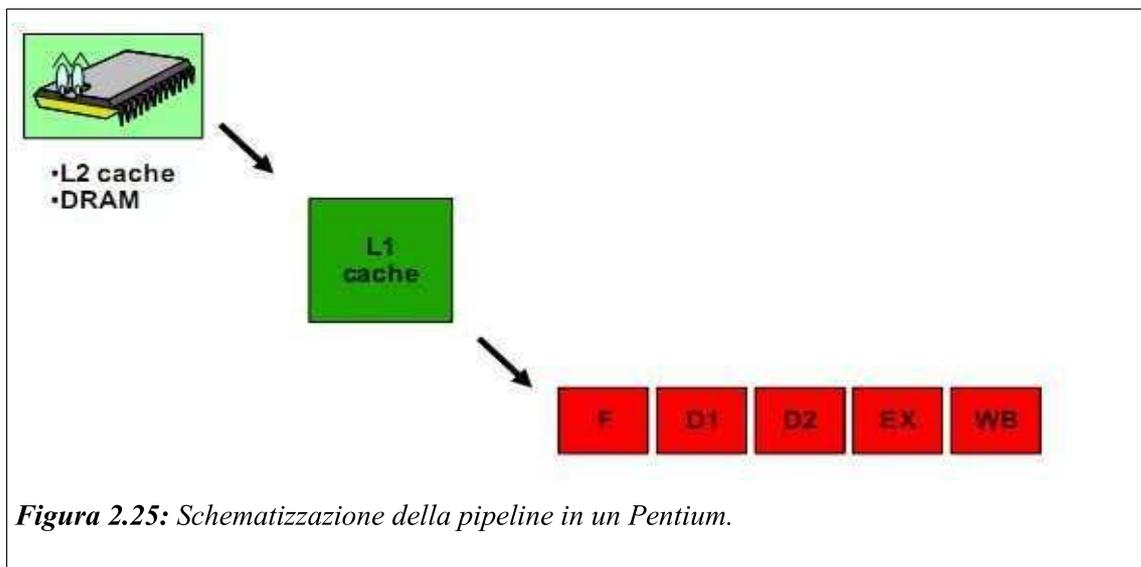
Occorre a questo punto tenere conto della frequenza di clock del processore, si deve quindi capire come sono correlati il tempo di clock e il tempo τ_{MAX} . I due tempi non sempre coincidono, infatti esistono stadi della pipeline formati da una macchina sequenziale (che richiede più periodi di clock per completare lo stadio).

Si assuma che τ_{MAX} sia dato dalla somma di M periodi di clock, si ha che i MIPS corrispondono all'inverso di τ_{MAX} ($1/\tau_{MAX}$); quindi sono proporzionali alla frequenza di clock, divisa per il numero di colpi di clock necessari per completare ogni stadio: f_{clk}/M .

Si consideri un processore RISC, nel quale M è uguale ad 1, ne deriva che la frequenza di clock determina anche i MIPS; se si considera un processore ad 1GHz, ogni ns viene completata un'istruzione e quindi si avrà come risultato 1000 MIPS, il che significa eseguire 1.000.000.000 di istruzioni al secondo.

2.7.3 Pipeline nel Pentium

Originariamente il primo Pentium aveva cinque stadi di pipeline, successivamente il Pentium IV aveva trenta stadi di pipeline, per poi tornare indietro a circa quindici stadi nelle architetture core duo, a causa dell'elevata complessità di gestione di un tale numero di stadi.



Consideriamo la pipeline a cinque stadi:

Fetch. La prima fase non esegue il fetch vero e proprio dell'istruzione, infatti non accede in memoria ma preleva l'istruzione dalla coda di prefetch; se così non fosse la fase di fetch durerebbe eccessivamente. Si stima che l'accesso a memoria duri $80ns$ mentre l'accesso alla coda di prefetch duri $2ns$; questo significa che la pipe deve stare ferma per 40 volte. Il vincolo che pone la fase di fetch sulla durata massima di ogni stadio è $\Delta T_F = 2ns$.

Decodifica del codice. In questa fase viene decodificato il codice operativo dell'istruzione e vengono effettuate alcune verifiche, in particolare vengono verificate tutte le compatibilità tra istruzioni, i così detti privilegi. A questo punto se la verifica è andata a buon fine si può proseguire con la seconda fase, altrimenti occorre scaricare la coda di prefetch e la pipeline. Il vincolo che

pone l'operazione di decodifica è legato ai circuiti di controllo, tipicamente combinatori. Si stima che la durata di questa operazione sia circa $\Delta T_{D1} = 2ns$.

Prelevamento degli operandi. Gli operandi possono essere o all'interno dei registri del processore oppure in memoria. Quindi il tempo ΔT_{D2} dipende da dove sono memorizzati gli operandi, nel caso siano memorizzati sui registri, il tempo ΔT_{D2} sarebbe poco rilevante; nel caso invece di accesso in memoria ΔT_{D2} dovrebbe essere pari a 80ns. Risulta fondamentale il bisogno di una L1 cache: una memoria piccola ma fisicamente vicina all'unità di controllo, nella quale vengono memorizzati i dati che hanno probabilità maggiore di essere richiesti dalle istruzioni in esecuzione. Quindi si determina il tempo ΔT_{D2} in base al tempo di accesso alla L1 cache: $\Delta T_{D2} = 3ns$.

Execution. La fase di execution è la fase in cui l'istruzione viene svolta. La sua durata dipende dal tipo di istruzione:

- Trasferimento. Sono operazioni relativamente veloci.
- Aritmetiche. Si dividono a loro volta in due categorie:
 - Fixed Point. Operazioni tra numeri interi, sono più lente delle operazioni di trasferimento, ma meno impegnative rispetto alle operazioni in virgola mobile.
 - Floating Point. Sono le operazioni di esecuzione più lunghe e impegnative, per poterle inserire in una struttura pipeline occorre dividerle in più stadi, in questo modo si avrebbero delle pipeline con due o più stadi di execution. Per le operazioni in floating point si utilizzano delle pipeline dedicate.

Se si suppone di inviare sulla pipeline solo le istruzioni che fanno riferimento ad operazioni di trasferimento o ad operazioni tra numeri interi, il tempo ΔT_{EX} è dato dalle operazioni matematiche che richiedono più tempo per essere eseguite: operazioni di shift, divisione e di moltiplicazione. Il tempo ΔT_{EX} è, quindi, legato al tempo di un'operazione di ALU: $\Delta T_{EX} = 2ns$.

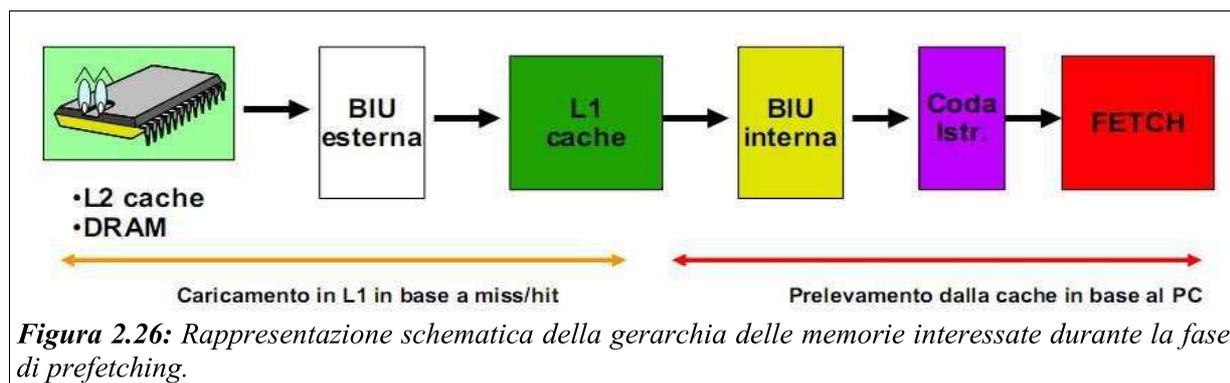
Write Back: La fase di write back è la fase in cui vengono scritti i risultati di un'istruzione, i quali possono essere memorizzati in dei registri o in memoria. Il ragionamento è analogo a quello fatto nella fase di prelevamento degli operandi, nel quale si sfruttava la presenza della memoria cache, che in questo caso viene utilizzata per scrivere i dati in una locazione di memoria vicina al chip. Perciò il tempo ΔT_{WB} è regolato dal tempo di accesso alla cache dati: $\Delta T_{WB} = 3ns$.

A questo punto si ha:

$$\Delta T_F = 2ns \quad \Delta T_{D1} = 2ns \quad \Delta T_{D2} = 3ns \quad \Delta T_{EX} = 2ns \quad \Delta T_{WB} = 3ns$$

Tra questi tempi si sceglie il tempo di durata maggiore: $\tau_{MAX} = 3ns$, tramite questo si sceglie la frequenza del clock di avanzamento: $1/\tau_{MAX} = 1/3 \text{ GHz}$ la quale deve essere correlata con la frequenza di clock del processore, supposta di 1 GHz . Si ha che la pipeline deve avanzare di un blocco ogni 3 colpi di clock.

2.7.4 Gestione della memoria

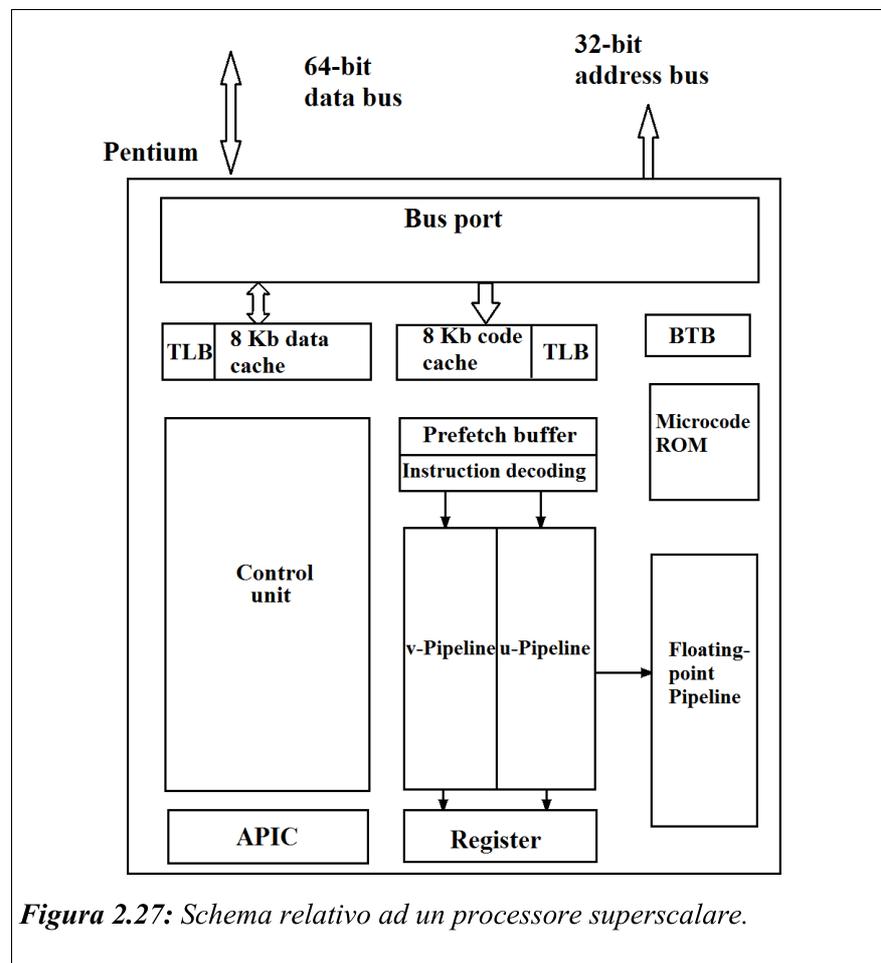


L'immagine 2.26 mostra il flusso completo durante il periodo di fetch di un'istruzione. E' indicato nel blocco rosso lo stadio di fetch, che preleva le istruzioni dal buffer che contiene la coda

di prefetch (blocco viola). Non appena si libera uno spazio nella coda delle istruzioni la BIU interna al chip del processore si occuperà di prelevare l'istruzione immediatamente successiva all'ultima estratta dalla L1 cache. Tale istruzione può essere presente all'interno della cache L1 o non esserlo. Nel caso sia presente viene caricata nella coda delle istruzioni (caso ideale), nel caso non sia presente si genera un cache miss; occorre quindi un'unità che si occupi di caricare nella cache L1 le istruzioni che devono essere recuperate dai livelli superiori di cache o dalla memoria; si occupa di questa operazione la BIU esterna.

Se il tempo di prelevamento di tutte le istruzioni presenti nella coda di prefetch risulta essere maggiore del tempo di aggiornamento della cache L1 non occorre bloccare la pipeline in attesa di ricevere istruzioni; in caso contrario, cioè se la coda delle istruzioni si svuota più velocemente del tempo che occorre per riempirla, si arriverà alla situazione in cui la coda delle istruzioni è vuota, in questo caso la pipeline si blocca. Si potrà continuare l'avanzamento degli stadi solo dopo che sarà presente un'ulteriore istruzione nella coda di prefetch.

2.8 Processore superscalare



Nell'immagine è mostrata l'architettura di un processore superscalare. In quest'architettura sono presenti due cache, una per il codice e una per i dati. Le cache sono separate tra loro per diverse ragioni: per semplicità realizzativa, in quanto la cache del codice è di sola lettura e può essere resa semplice in termini strutturali; ma la ragione principale è che tenendo separate le due cache si possono effettuare operazioni parallele, essenziali in un architettura pipeline. Con questo tipo di architettura delle memorie cache possono essere eseguite contemporaneamente le operazioni di fetch e write back, la prima legge l'istruzione dalla *code cache*, la seconda scrive il risultato dell'istruzione precedente sulla *data cache*. Direttamente connesso alla cache di codice si può

notare il buffer di prefetch e la decodifica dell'istruzione, dopo la quale sono presenti tre pipeline, due delle quali v e u eseguono operazioni simili in parallelo, mentre la terza esegue le operazioni su valori floating point. L'operazione di decodifica viene fatta prima di inviare le istruzioni nelle pipeline in modo da scegliere la pipeline più adeguata a svolgere l'operazione richiesta.

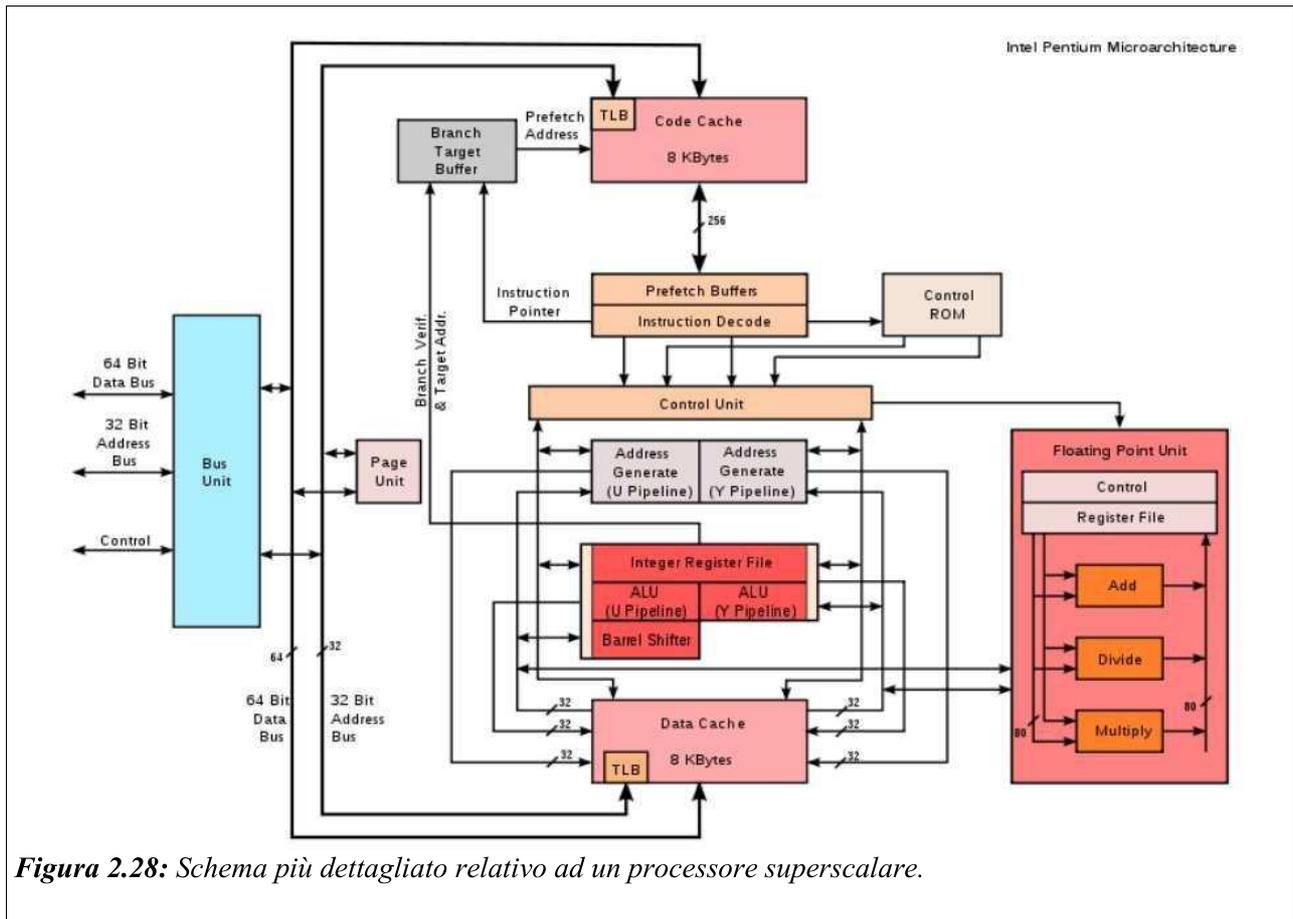


Figura 2.28: Schema più dettagliato relativo ad un processore superscalare.

Nell'immagine 2.28, si nota che le due pipeline parallele (blocchi rossi) sono asimmetriche, il che significa che alcune istruzioni possono essere eseguite solo in una delle due pipeline; questo deriva dal fatto che il numero di transistor era limitato e le due pipe non potevano essere fatte completamente equivalenti. Inoltre si può notare con maggiore dettaglio la struttura della pipeline dedicata alle operazioni in virgola mobile, nella quale sono presenti blocchi hardware dedicati al

calcolo di operazioni di moltiplicazione, divisione e addizione tra float. Si noti che il numero di stadi di una pipeline dedicata alle operazioni in floating point sono 8 contro i 5 che occorrono alle altre due pipe, questo è dovuto al fatto che la fase di execution della pipeline dedicata si divide in più blocchi di calcolo.

L'unità di controllo si occupa di analizzare le istruzioni tre per volta per verificare se sono parallelizzabili, in quel caso, molto raro, le invia contemporaneamente su tre pipe differenti.

2.8.1 Valutazione delle prestazioni

Nel caso di P pipeline operanti in parallelo il numero di MIPS corrisponde a:

$$MIPS = \left(\frac{1}{\tau_{MAX}}\right) \cdot P = \left(\frac{f_{clk}}{M}\right) \cdot P$$

Dove P equivale al numero di pipeline che operano in parallelo e M al numero di colpi di clock richiesti per eseguire le operazioni in uno stadio.

Se si considera come frequenza di clock 1 GHz , con $M=1$, e $P=5$ si ha un MIPS pari a 5000. Che corrisponde a 5 miliardi di istruzioni al secondo.

Occorre tener presente che la valutazione delle prestazioni è fatta sul caso migliore in cui le istruzioni sono tutte parallelizzabili, non si verificano mai cache miss e la pipe non va mai in stallo. Queste situazioni sono puramente ideali e la probabilità che un evento del genere si verifichi è estremamente bassa.

2.8.2 Architettura superscalare di ultima generazione

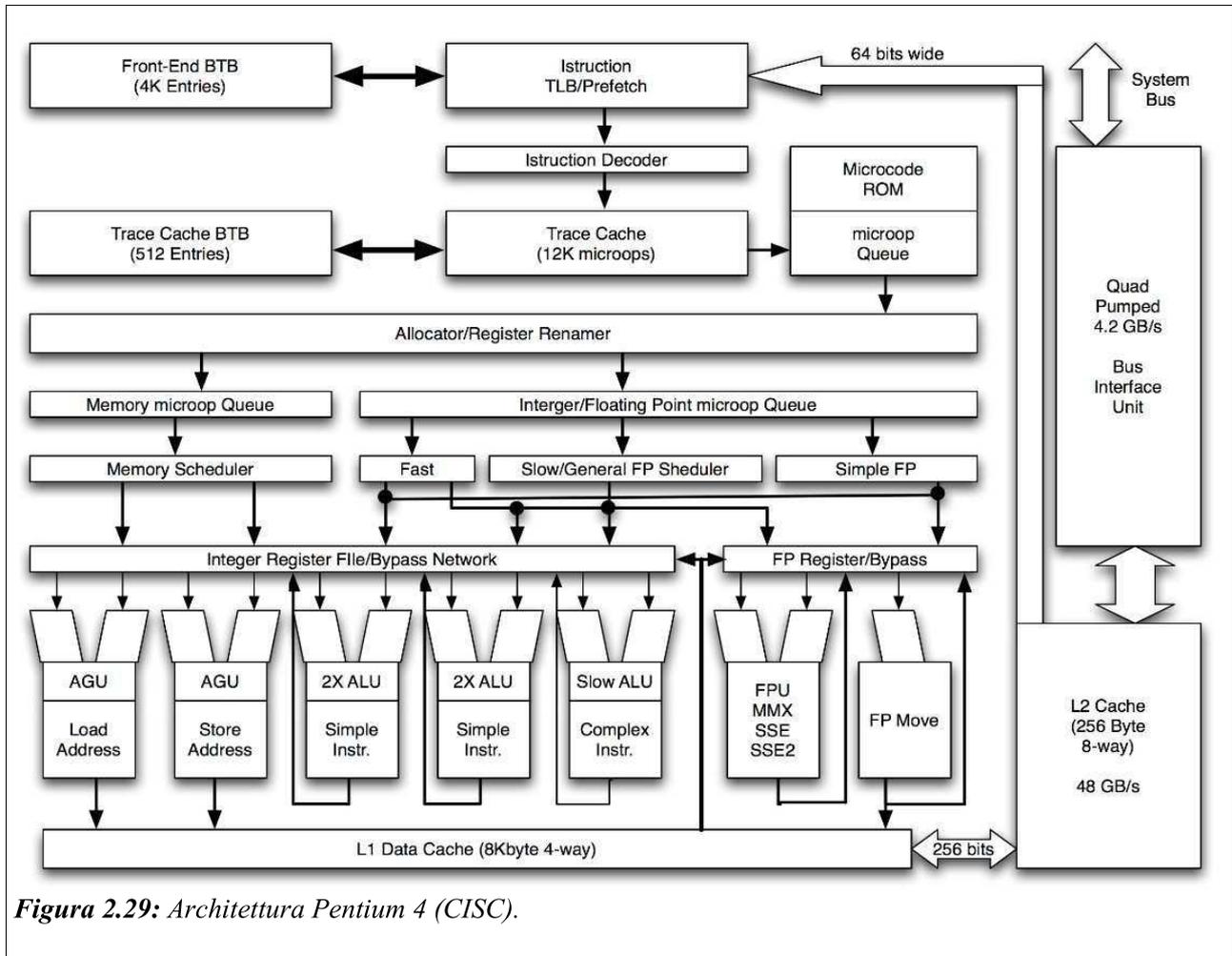


Figura 2.29: Architettura Pentium 4 (CISC).

Rispetto alle architetture analizzate in precedenza è articolata in un numero molto maggiore di pipeline e le unità operative sono molto più parallelizzate. In questa architettura le singole istruzioni a livello assembler vengono “esplose” in una sequenza ordinata di micro-operazioni che variano a seconda dell'istruzione che viene considerata, dal momento che si hanno più istruzioni assembler che lavorano in parallelo, le micro-operazioni vengono fuse insieme e il parallelismo non viene fatto a livello di istruzione assembler ma a livello di micro-operazione; questo permette di avere un livello più basso di parallelismo e di avere maggiore facilità nel parallelizzare. Al termine

dell'esecuzione di una micro-operazione occorre ricordarsi a quale istruzione essa apparteneva per poterla ricomporre, operazione da non sottovalutare in quanto a complessità; inoltre poiché ogni istruzione viene divisa in una serie di micro-operazioni occorre definire un numero maggiore di registri. Per fare ciò il registro AX, ad esempio, viene duplicato in una serie di istanze in modo tale che le micro-operazioni possano essere eseguite sulle varie istanze al fine di poterle parallelizzare efficacemente. A livello hardware ci saranno, quindi, una molteplicità di registri che a livello assembler saranno visti come un registro unico; questo crea un particolare effetto di complicazione la cui gestione è tutt'altro che banale in termini di unità di controllo.

2.9 I/O

Programmare un dispositivo significa scrivere un'opportuna sequenza di bit all'interno dei suoi registri. Questi registri, a livello assembler, possono essere visti come indirizzi a cui corrisponde un'interfaccia.

Dal punto di vista hardware, la gestione degli I/O può essere organizzata in due modalità fondamentali: memory mapped I/O e isolated I/O.

2.9.1 Memory Mapped I/O

Dal punto di vista del processore gestire una cella di memoria o gestire un dispositivo di I/O equivale a gestire un indirizzo. Dal punto di vista hardware questo viene realizzato tramite un decodificatore che preleva l'indirizzo, lo decodifica e invia al chip di memoria, o al registro di interfaccia del dispositivo di I/O, un segnale di abilitazione. Perciò si può riservare una parte della

memoria sulla quale si possono “mappare” gli indirizzi delle interfacce di I/O, così quando si scriverà su di una cella di memoria mappata per gli I/O, non si scriverà effettivamente su quella cella, ma sul dispositivo di I/O mappato su di essa.

Per poter realizzare tale architettura, occorre abilitare il segnale MEM, anche nel caso di I/O, questo implica l'utilizzo delle istruzioni tipiche della memoria (MOV) anche quando ci si riferisce a dispositivi di I/O.

2.9.2 Isolated I/O

Nel caso di Isolated I/O, gli indirizzi delle interfacce di I/O sono mappati in una zona di memoria separata dalla memoria RAM. Per poter differenziare tali zone occorre abilitare un segnale di I/O quando ci si vuole riferire ai dispositivi di Input Output; tale operazione è effettuata automaticamente dal processore a seconda dalle istruzioni che vengono decodificate, esistono infatti istruzioni specifiche per le operazioni di I/O: la IN e la OUT. In pratica quando un 8086 deve eseguire un'istruzione IN o OUT metterà a zero la linea di controllo $M/\bar{I}\bar{O}$, mentre la terrà a 1 per tutte le altre istruzioni, che devono accedere alla memoria ordinaria.

2.9.3 Istruzioni di Input e Output

La forma più tipica delle istruzioni di I/O dell'8086 è la seguente:

IN AL, DX ; input data from I/O port

OUT DX , AL; output data to I/O port

oppure

IN AL , <Indirizzo immediato minore di 256>

OUT <Indirizzo immediato minore di 256> , AL

Il dato di queste istruzioni deve essere sempre messo in AL, mentre l'indirizzo va in DX se si vuole usare l'indirizzamento indiretto. Si può usare anche un indirizzamento immediato, ma il valore dell'indirizzo di I/O deve essere sempre minore di 256 (numero di 8 bit).

IN (input) è l'istruzione di ingresso: il dato viene trasferito dalla porta di I/O, il cui indirizzo è specificato in DX, al registro AL.

OUT (output) è l'istruzione di uscita: il dato contenuto in AL viene copiato alla porta di indirizzo indicato da DX.

Si noti che in queste istruzioni gli unici registri che si possono usare sono AL e DX. DX contiene l'indirizzo a 16 bit non segmentato e AL il dato da leggere o scrivere.

2.9.4 Vantaggi e svantaggi

Nel caso di memory mapped si ha il vantaggio di usare le istruzioni di memoria per gestire gli I/O, le quali sono più potenti delle istruzioni di I/O tradizionali. Come svantaggio si deve riservare una porzione di memoria all'interno della RAM. Questa quindi non sarà lineare poiché si deve utilizzare una parte di essa per la gestione degli I/O; perciò nel caso in cui occorra aumentare la memoria in funzione delle esigenze (come accade nei server) tale operazione non sarà possibile perché un determinato spazio di essa deve rimanere riservato.

Nel caso di Isolated I/O, si ha il vantaggio che lo spazio di indirizzamento della memoria RAM è completamente a disposizione e può essere ampliato anche in situazioni di accrescimento successivo. Come svantaggio si è costretti ad utilizzare per gestire gli I/O delle istruzioni dedicate,

più semplici e semanticamente meno significative di quelle della gestione della memoria.

Nei sistemi dedicati (telefoni cellulari), in cui non si ha la necessità di espandere la memoria si può, per semplicità e per potenza di scrittura scegliere la soluzione memory mapped. Nei sistemi general purpose (Personal Computer), dove si ha il vincolo di un'estendibilità della memoria si è forzati ad andare verso soluzioni di Isolated I/O.

Nel caso x86 è possibile adottare entrambe le soluzioni, perché non dipendono dal processore ma dall'hardware di decodifica, esterno ad esso. Nel caso si usi Isolated I/O lo spazio di indirizzamento è di 64Kbyte perciò un indirizzo riferito alle interfacce di I/O è di soli 16 bit e non è segmentato; si noti che con 16 bit si possono gestire 64kIO, un numero molto elevato, tanto è vero che le CPU x86 più recenti usano ancora indirizzi di I/O a 16 bit.

2.10 Registri speciali

2.10.1 FPU

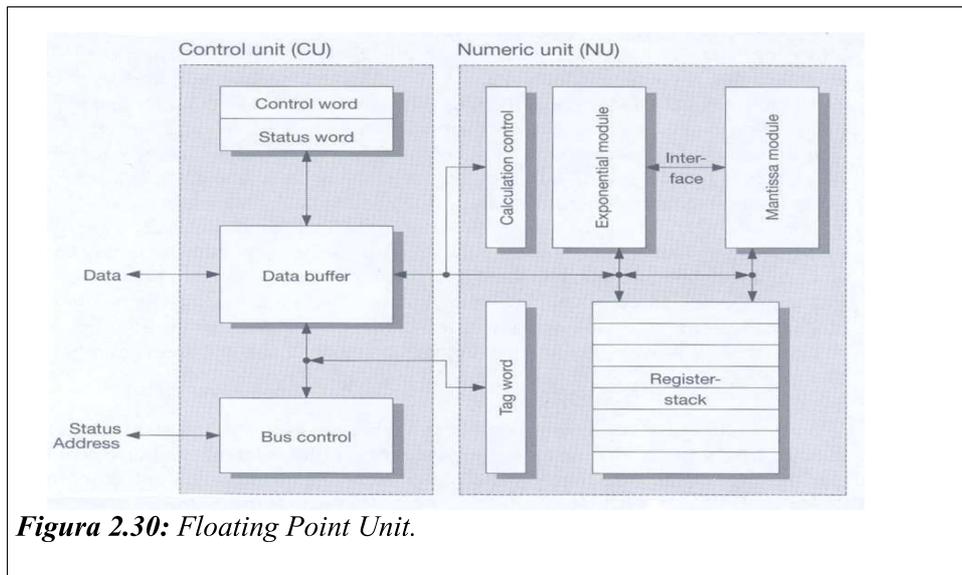
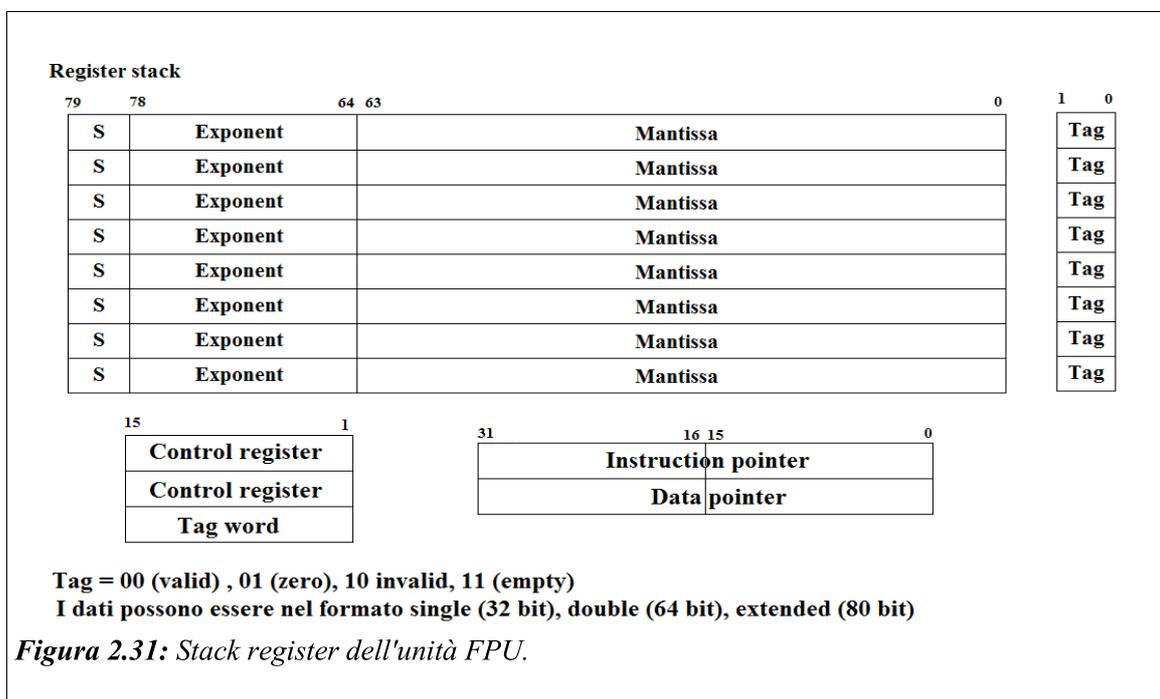


Figura 2.30: Floating Point Unit.

La Floating Point Unit ha dei registri che non fanno riferimento ai registri standard del sistema 8086, ma fa riferimento ad uno stack di registri, che non possono essere acceduti singolarmente ma solo mediante il top dello stack. La FPU segue lo standard IEEE754².

2.10.1.1 Registri dati



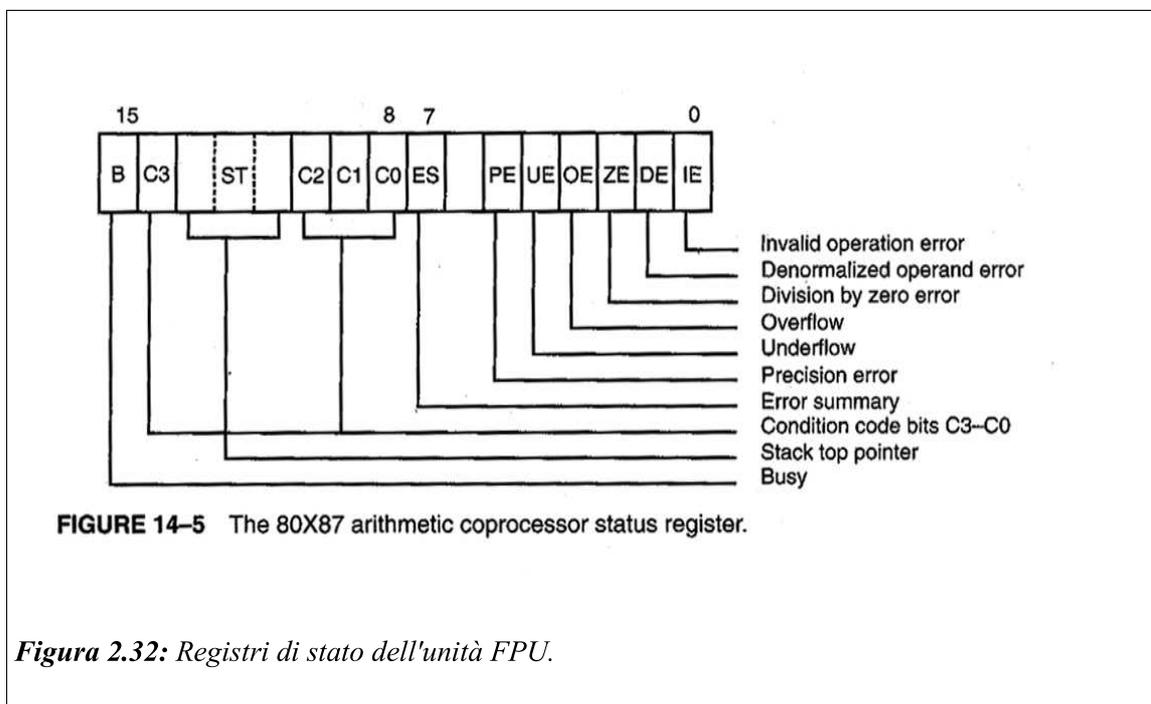
L'unità FPU ha 8 registri di dato, ciascuno dei quali contiene il segno, l'esponente e la mantissa del numero che vi è memorizzato; associato a ciascun registro vi è un TAG che indica la validità o

² Lo standard IEEE per il calcolo in virgola mobile (IEEE 754) (ufficialmente: IEEE Standard for Binary Floating-Point Arithmetic) è lo standard più diffuso nel campo del calcolo automatico. Questo standard definisce il formato per la rappresentazione dei numeri in virgola mobile, ed un set di operazioni effettuabili su questi. Specifica inoltre quattro metodi di arrotondamento e ne descrive cinque eccezioni.

la disponibilità del registro stesso. I registri hardware sono registri a 80 bit, di cui 64 sono dedicati alla mantissa, 15 all'esponente e 1 al segno.

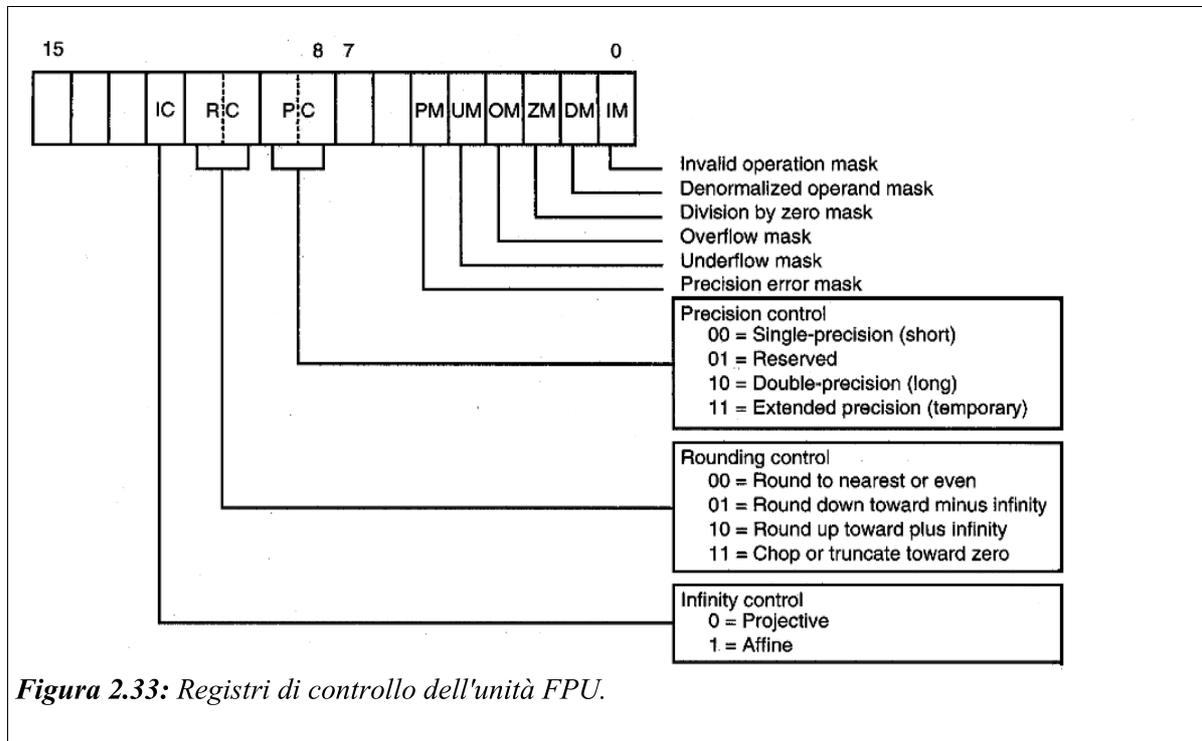
Si può programmare la FPU per poter operare su un formato singolo o un formato doppio, questi due formati sono i due compatibili con lo standard IEEE.

2.10.1.2 Registri stato



Il registro di stato contiene una serie di flag di controllo, tipici delle operazioni matematiche, riportati in figura 2.32.

2.10.1.3 Registri di controllo



Il registro di controllo permette di identificare le modalità con cui si vuole operare, come la singola o la doppia precisione (a 32 o 64 bit); oppure il tipo di arrotondamento che si vuole utilizzare (troncamento o arrotondamento al più vicino). Questi elementi devono tutti essere impostati in fase di programmazione dell'unità FPU.

2.10.2 MMX (Multimedia eXtension Mutiple Math o Matrix Math eXtension)

Con l'avvento della grafica è nato il problema della manipolazione dei pixel, che prevede sostanzialmente di lavorare con delle maschere su una sequenza molto lunga di bit. Questo tipo di operazioni portano ad una sorta di parallelismo operativo sui dati.

Si supponga che un pixel sia codificato su 3 bit (R-G-B), e di voler eliminare la componente blu da una serie di pixel; per poterlo fare occorre effettuare un'operazione di AND con una maschera del tipo 1-1-0. Questa operazione deve essere ripetuta per il numero totale di pixel, che può essere molto elevato; per velocizzare quest'operazione si utilizzano registri molto grandi chiamati XMMi, su cui vengono memorizzati una serie di pixel, sui quali verrà applicata contemporaneamente la maschera 1-1-0.

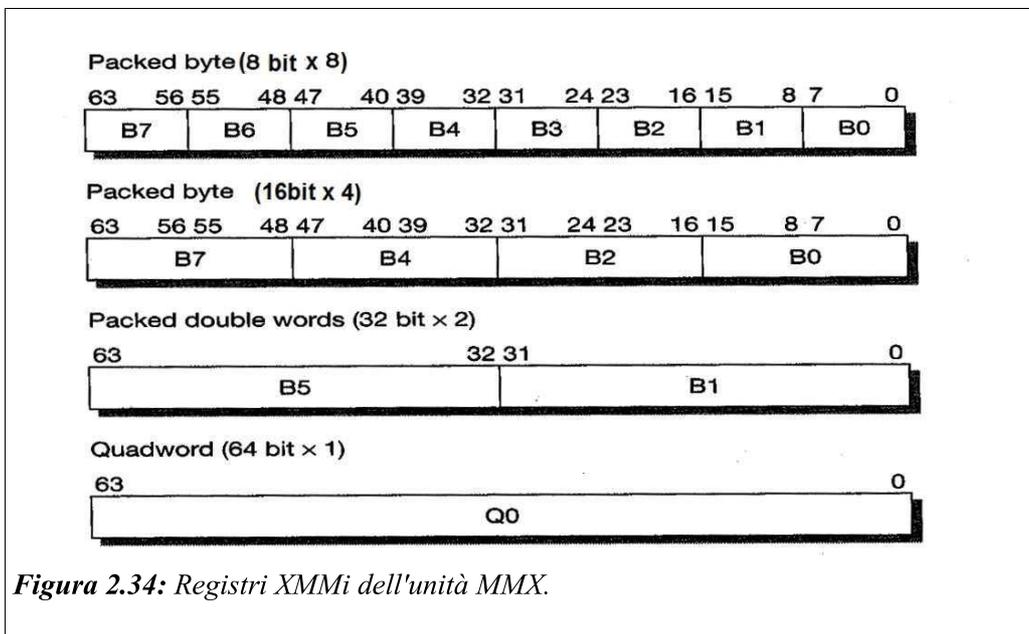
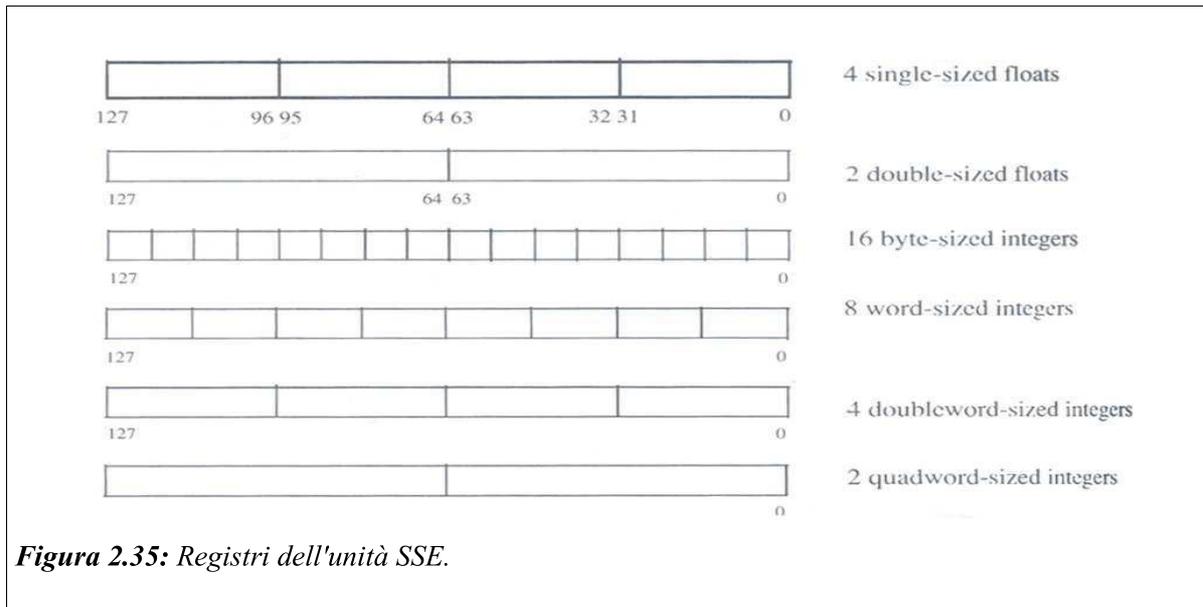


Figura 2.34: Registri XMMi dell'unità MMX.

All'interno dell'unità MMX sono presenti 8 registri XMMi, grandi 64 bit ciascuno. Questi registri possono essere organizzati in termini di byte, word, dword o qword, come mostrato in figura 2.34.

2.10.3 SSE (Streaming SIMD Extensions)

Questa tecnologia applicata inizialmente alla grafica ha trovato grande successo in tutte le operazioni di multimedia avanzata, in particolare sull'operazione di streaming, non solo sui numeri interi ma anche sui numeri in floating point.



L'unità SSE può memorizzare dati più complessi perché i suoi registri sono più lunghi: 128 bit. Si possono quindi avere codifiche sui dati in floating point sia in singola che in doppia precisione.

Inoltre l'unità SSE permette di eseguire operazioni sui vettori con un notevole risparmio di tempo rispetto alle architetture tradizionali: si può memorizzare un vettore di byte lungo fino a 16 posizioni in un unico registro, e nel caso lo si voglia sommare con un altro vettore presente su di un altro registro basta semplicemente eseguire l'operazione di ADDPS, che permette di effettuare contemporaneamente la somma di tutte le celle dei due vettori.

Le attuali architetture permettono non solo di realizzare architetture superscalari, ma anche architetture vettoriali sulla base di modelli SIMD (Single instruction multiple data).

3. Bus di sistema

3.1 Definizioni generali

Volendo dare una definizione possiamo dire che nei sistemi elettronici e nei computer in particolare, il bus è un canale che permette a dispositivi periferici e componenti del sistema di "dialogare" tra loro. Diversamente dalle connessioni punto-punto un solo bus può collegare tra loro più dispositivi.

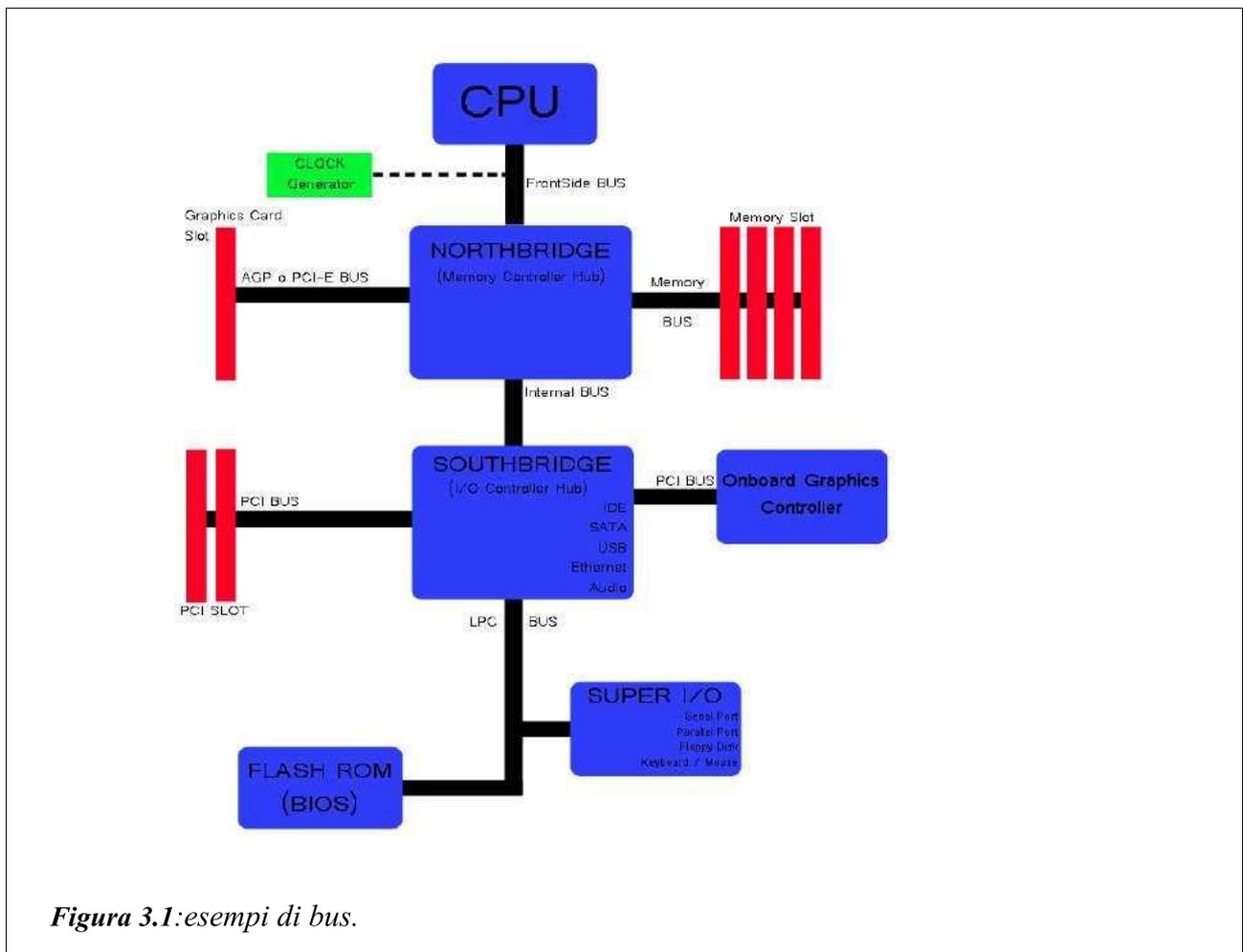


Figura 3.1: esempi di bus.

Si nota dalla figura 3.1 che non è presente un solo livello di bus ma ve ne sono molteplici che

hanno il compito di andare a collegare i vari livelli del calcolatore partendo dai periferici di I/O finendo alla CPU. In particolare sono rappresentati:

- Bus LPC: un bus la cui velocità massima è di 6.67 MBps e il cui compito è quello di andare a collegare le periferiche di I/O e la Flash Rom del Bios con il SouthBridge, una sorta di *arbiter* il cui compito è quello di andare a gestire i collegamenti con le periferiche con le quali instaura delle connettività quasi punto a punto.
- Bus PCI: un bus da 132 MBps che ha il compito di andare a collegare il SouthBridge con i controller della grafica.
- Internal Bus: bus il cui compito è di andare a collegare il SouthBridge con il NorthBridge che si occupa della gestione efficace della memoria, contiene il cache control (attraverso opportune transazioni sul bus si occupa dei miss e degli hit, quindi del caricamento e dello scarico in memoria cache delle porzioni di memoria principale) e non gestisce gli I/O.
- System Bus: un bus da 528 MBps, solitamente detto AGP, che ha il compito di collegare il NorthBridge con la scheda grafica.
- Memory Bus: un bus da 528 MBps il cui compito è quello di andare a collegare il NorthBridge con la memoria.
- Front Side Bus: un bus il cui compito è quello di andare a collegare la Cpu con il NorthBridge.

Vediamo che è stata precedentemente usata la scritta MBps (Mega Byte per secondo) tenendo conto che in generale si indica con tale unità di misura la capacità di trasferimento del bus.

In oltre, facendo riferimento alle architetture 80x86, possiamo suddividere i bus in tre tipologie differenti che sono:

- Bus sincroni: sono quei bus caratterizzati dall'aver il clock, segnale usato per la temporizzazione e il cui numero di periodi è fissato per ogni ciclo.
- Bus asincroni: sono quei bus sprovvisti di clock che viene sostituito da dei segnali di handshaking.
- Bus semisincroni: sono quei bus provvisti di clock il cui numero di periodi non è fissato per ogni ciclo ma dipende dal tipo di ciclo stesso.

3.2 Cicli di bus

Un ciclo di bus è la sequenza di eventi attraverso la quale la CPU comunica con la memoria, con un dispositivo di I/O o con l'Interrupt Controller. Inoltre esso può essere definito come il tempo attraverso cui si realizza completamente una transazione tra CPU e la memoria o i dispositivi di I/O. Considerando l'architettura dei processori 80x86 un ciclo di bus è costituito da 2 o 4 colpi del clock del bus stesso. Il ciclo di bus è composto da 4 fasi denominate T1, T2, T3, T4. Durante la fase T1, l'indirizzo viene scritto sull'address bus, mentre durante le fasi T2, T3 e T4 viene messo il dato sul data bus. Se la CPU non deve accedere all'esterno, i segnali di controllo del bus sono inattivi ed i relativi piedini sono in alta impedenza. Considerando, quindi, che nelle architetture 80x86 ogni ciclo di bus si compone di 2 colpi di clock, è possibile calcolare le prestazioni di un bus attraverso la formula:

$$\text{parallelismo del processore} = \frac{\text{frequenza del bus}}{2 \text{ colpi di clock}}$$

Supponendo quindi di avere una macchina Pentium con DBUS a 64 bit e 100MHz si ha:

$$\frac{64}{8} \frac{100\text{M}}{2} = 400\text{MBps}$$

essendo 2 i periodi di clock per ciclo di bus.

Attualmente, per aumentare il throughput, non viene incrementata la frequenza di clock ma si utilizza la tecnologia *double data rate*, in cui i trasferimenti vengono effettuati sia sul fronte di salita che su quello di discesa del clock. I *global data rate* sono dei meccanismi che permettono di eseguire o adottare questa tecnica. Perciò non si parla più di bit/s ma di trasferimenti al secondo (MT/s o GT/s) per rendere l'indice di prestazioni indipendente dal timing e indipendente dal parallelismo. Se si ha, ad esempio, una frequenza di clock a 100 MHz double data rate, equivale ad operare ad una frequenza doppia poichè si va ad interagire sia sul fronte di salita che sul fronte di discesa del clock stesso; in tal caso si ottengono 200 MT/s. Se poi si ha un bus da 64 bit (quindi 8 byte) si ottengono $200 \cdot 8$ MT/s; se si ha un bus da 1 byte si otterrà $200 \cdot 1$ MT/s e così via.

Esistono quattro tipologie di cicli di bus:

ciclo di lettura della memoria

ciclo di scrittura della memoria

ciclo di lettura da I/O

ciclo di scrittura su I/O

Per analizzare che cosa accada in un ciclo di bus, consideriamo il seguente esempio:

MOV DS[BX], AH.

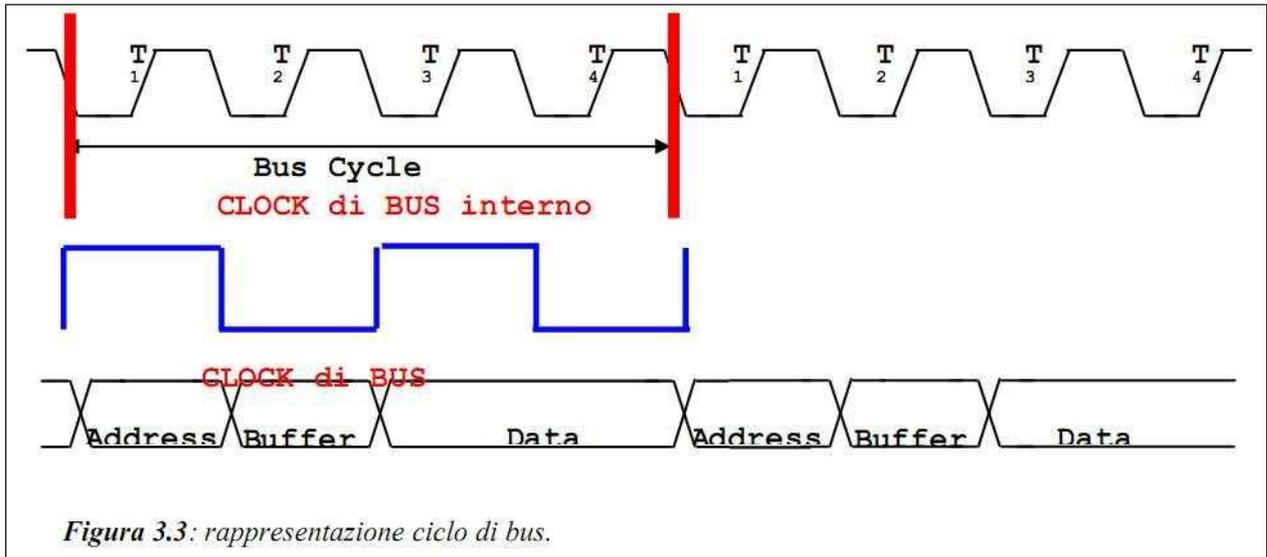
In questo caso avremo un'istruzione di 3 byte: un byte per il codice operativo, due byte contenenti l'indirizzo (offset). Supponendo di avere un DBUS a 16 bit, l'esecuzione dell'istruzione richiede due cicli di bus di "lettura memoria" (fetch) e un ciclo di bus di "scrittura memoria" (execution).



Figura 3.2: cicli di bus richiesti.

3. Bus di sistema

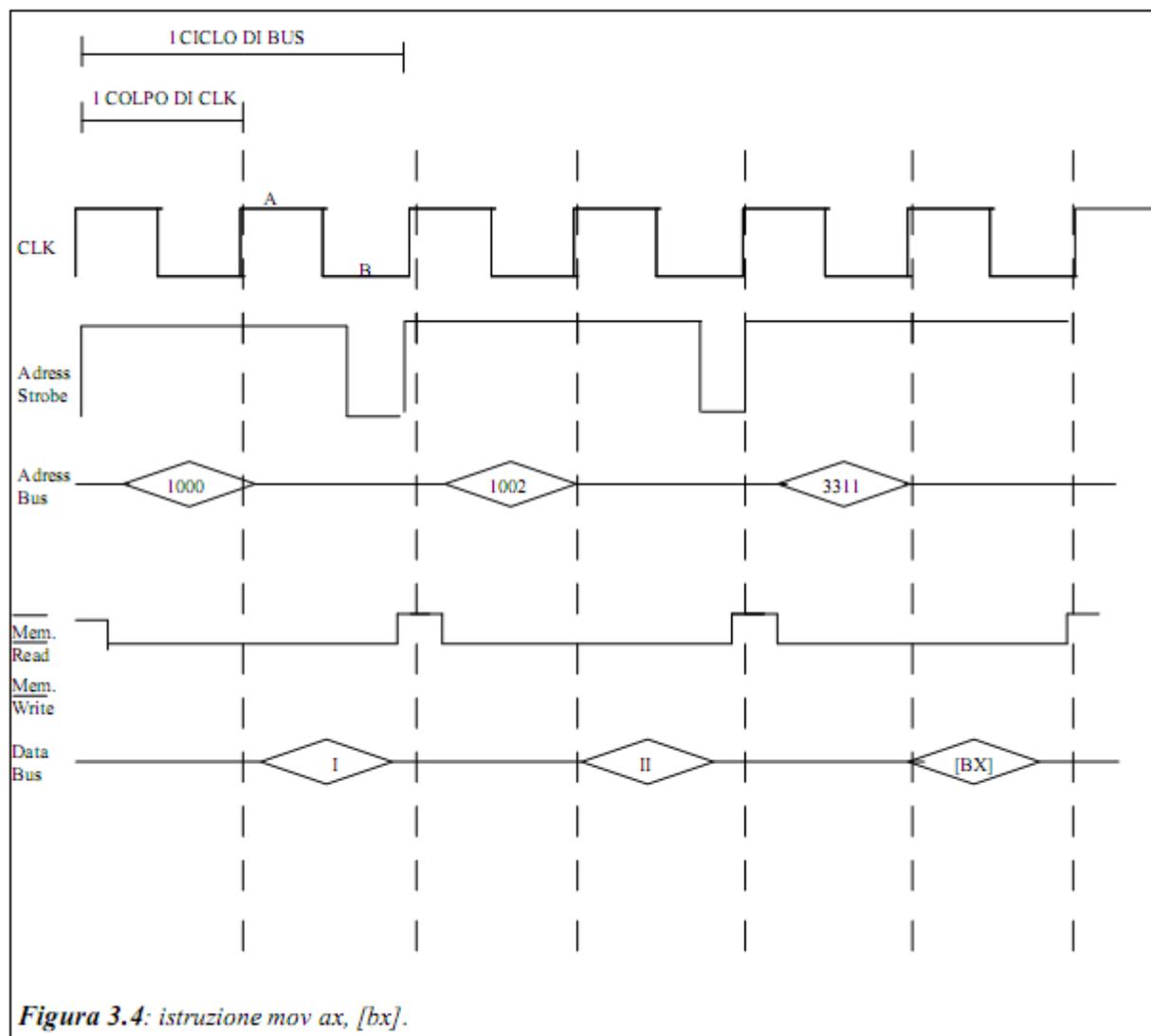
Durante il periodo T1 verranno messi sul bus gli indirizzi che, durante T2, verranno successivamente salvati nel buffer. Infine durante il periodo T3 e T4 verranno messi sul bus i dati veri e propri (vedi fig. 3.3).



Oltre alle differenti tipologie di bus prima enunciate, esistono altre due tipologie:

1. Cicli di Idle: vengono inseriti dalla CPU quando essa non necessita di nuovi dati oppure quando la coda interna delle istruzioni è piena, e non può essere eseguita alcuna fase di prefetch.
2. Cicli di wait: Se la memoria non è sufficientemente veloce, lo segnala alla CPU, e questa inserisce tra T3 e T4 una serie di stati di attesa (*wait states*) sinchè la memoria non risponde. Per comunicare all'80x86 la necessità di uno o più cicli di wait, la memoria esterna invia un segnale sul pin READY.

Si consideri ora il caso in cui si ha l'istruzione MOV AX, [BX] supponendo che CS:IP=PC=1000 e che DS:BX=3311.



Un segnale di Address Strobe viene attivato ogni volta che comincia un ciclo di bus (la sua funzione è infatti quella di avvisare la CPU dell'inizio di un ciclo di bus). Dopo di che il segnale attivo basso di memory read viene attivato (quindi passa da 1 a 0) e sull'address bus viene scritto l'indirizzo 1000 (CS:IP) che corrisponde all'indirizzo del primo byte dell'istruzione. Successivamente vengono scritti sul DataBus i dati relativi a questo primo byte dell'istruzione (che in fig. 3.4 è rappresentato con I). Si conclude a questo punto il primo ciclo di bus che sarà seguito da altri due (l'istruzione infatti è di 3 byte). È importante notare che nell'ultimo ciclo (quello in cui

fisicamente viene scritto in memoria il dato, quindi il ciclo di esecuzione vero e proprio del comando) sull'address bus viene scritto l'indirizzo 3311 (DS:BX), sul data bus il contenuto in memoria di [BX] e contemporaneamente attivato il segnale attivo baso memory write per segnalare l'inizio della scrittura in memoria. Infine dobbiamo dire che nel caso in cui la memoria non sia pronta essa lo segnala alla CPU, con il segnale di Ready, che inserisce tra A e B tanti cicli di Wait quanti necessari affinché essa sia pronta. In generale varrà la relazione:

$$(t. \text{ di accesso in memoria}) \leq (t. \text{ di accesso alla CPU}) + n \cdot (tw)$$

dove n è il numero di cicli di wait inseriti dalla CPU mentre tw è il tempo necessario per un ciclo di wait che nelle architetture 80x86 è pari ad un colpo di clock.

3.3 Cicli di bus nel Pentium

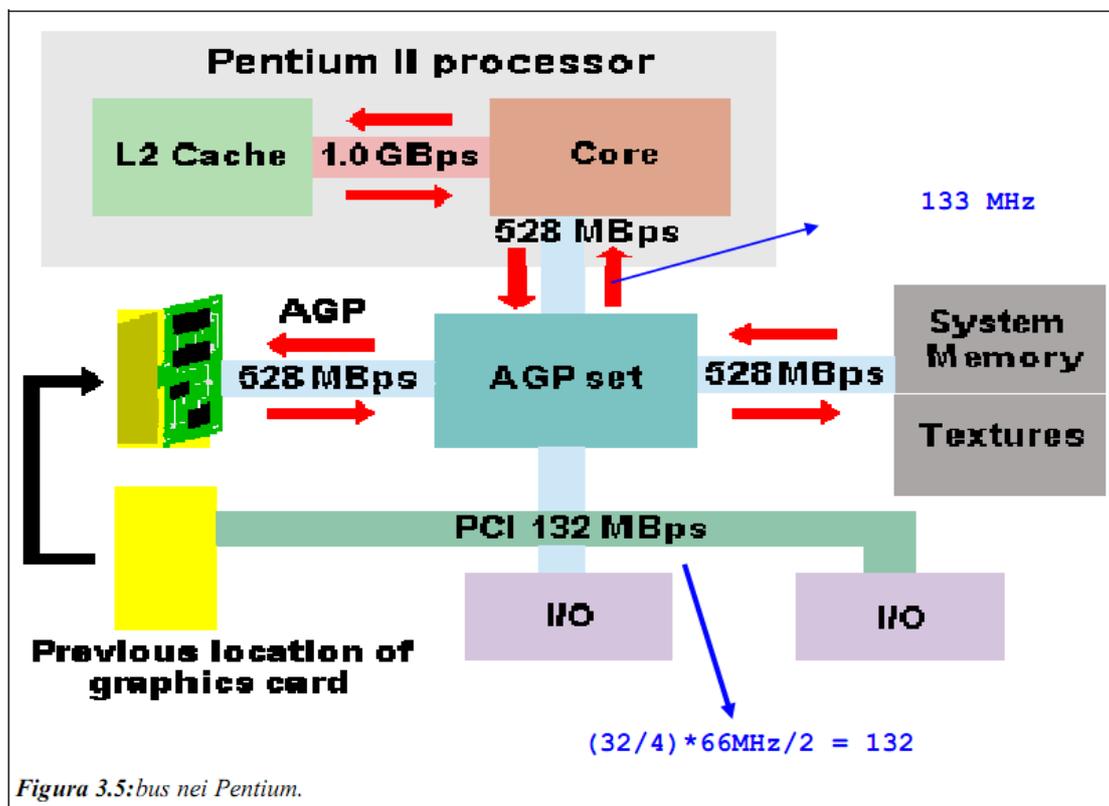


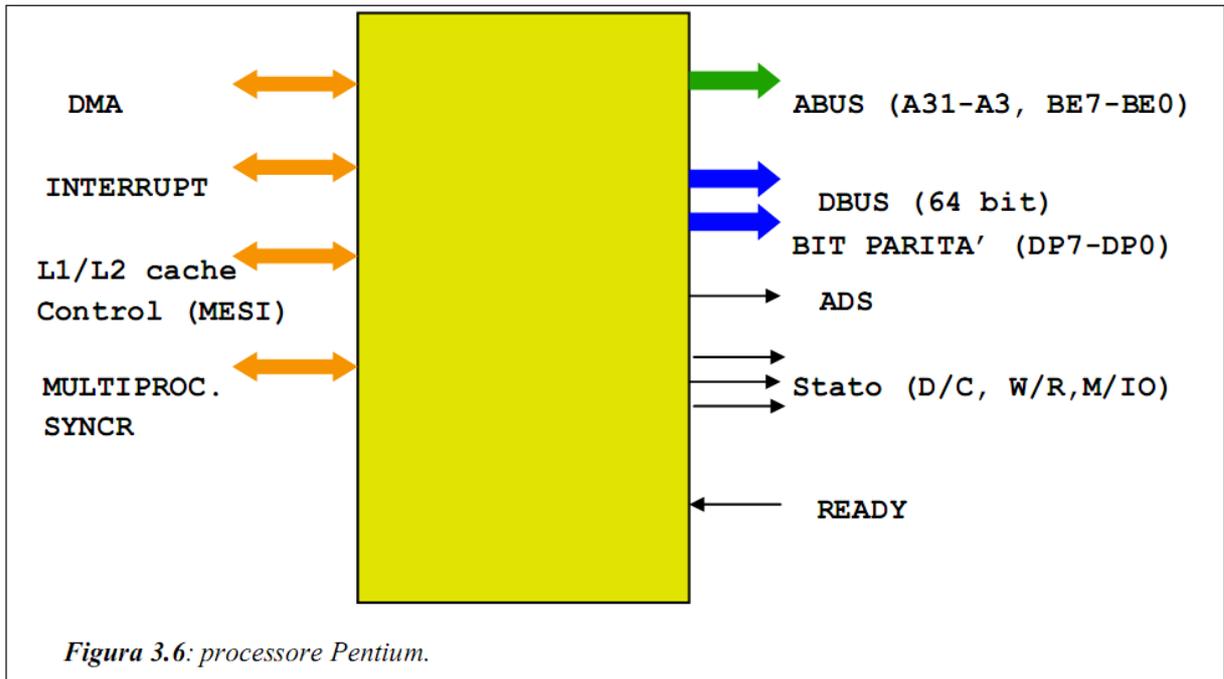
Figura 3.5: bus nei Pentium.

La figura 3.5 fa riferimento ad una tipica rappresentazione dei bus nelle architetture Pentium. Anche in questo caso è possibile notare che la conformazione è simile a quella rappresentata in Figura 3.1 escluso che, in questo caso, vi è la presenza di un bus da 1GBps che collega la CPU con la cache (all'interno della quale vengono memorizzati quei dati che si suppone verranno riutilizzati a breve e a cui l'accesso deve essere molto veloce).

Sono presenti, inoltre, molteplici tipologie di bus caratterizzate da diverse velocità. Ogni volta che la CPU necessita di un dato, ad esempio dalle periferiche di I/O, è necessario scendere di livello (in questo caso passando per i due bridge) con una conseguente riduzione della velocità. Nonostante la CPU disponga di un bus molto veloce è costretta ad aspettare, in questo caso, che i dati attraversino il bus LPC; durante questa attesa la CPU non può fare nessun'altra operazione.

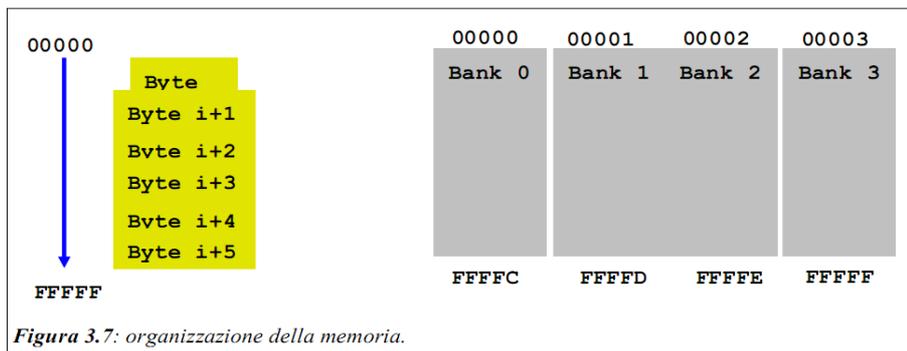
Una possibile soluzione a questo problema potrebbe essere quella di utilizzare dei bus con velocità uguali. Tale soluzione non può essere adottata per due motivi fondamentali:

1. Architettura: un bus deve essere tanto veloce quanto lo sono i dispositivi ad esso collegati (se avessimo un bus molto veloce esso dovrebbe attendere comunque un determinato tempo affinché il dispositivo gli fornisca i dati);
2. Fisica realizzabilità del bus: la lunghezza d'onda è legata alla frequenza attraverso la relazione $\lambda = c / f$. Andare ad aumentare la frequenza significherebbe andare a diminuire la lunghezza d'onda il che vuol dire andare a realizzare dei fili più lunghi tra cui si andrebbero, però, a creare delle capacità parassite molto elevate. Il rischio sarebbe quindi che il segnale si propaghi non più attraverso i fili ma bensì attraverso l'etere (meccanismo che non è gestibile dai dispositivi attuali).

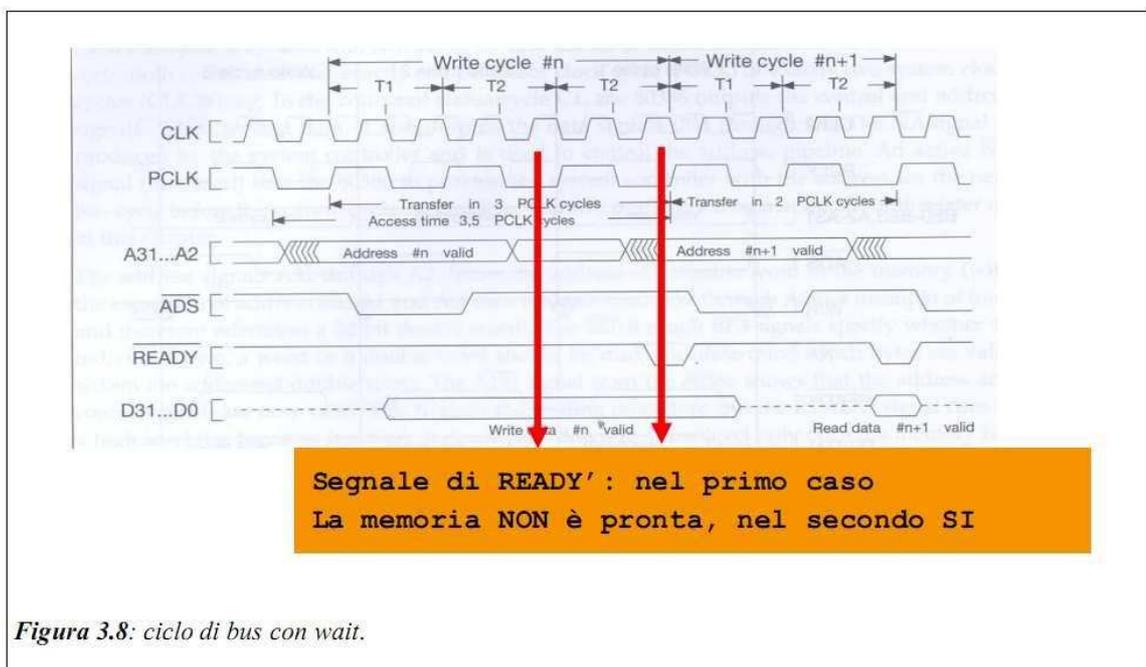


Nella figura precedente vediamo che sono stati rappresentati i seguenti segnali:

- Adress bus: esso serve per il trasporto degli indirizzi. Gli ultimi 29 bit sono usati per scegliere da quale riga andare a leggere mentre al posto dei primi 3 vengono inseriti altri 8 bit detti segnali BEi* (*Bank Enable*) che selezionano il/i byte interessati al trasferimento determinato da una data istruzione. Da un punto di vista logico la memoria è vista come una sequenza di byte ad indirizzi crescenti, mentre la memoria fisica è organizzata in banchi. I segnali BEi* permettono di selezionare il banco fisico della memoria cui corrisponde un dato indirizzo.



- Data Bus: bus utilizzato per il trasferimento dei dati
- Bit di parità: essi vengono usati per il controllo degli errori. Sono 8 (1 per ogni byte essendo il Data Bus da 64 bit=8 byte) e seguono il DataBus dopo il suo invio.
- Address Strobe: utilizzato per segnalare alla CPU l'inizio di un ciclo di bus
- Stato: dei segnali che servono per abilitare la lettura o la scrittura della memoria e dell'I/O
- Ready: segnale attraverso cui la memoria comunica alla CPU che è pronta per effettuare le varie operazioni (esempio di ciclo di bus con segnale di ready mostrato in figura)



- DMA: sono i segnali di *hold* e *holda*. Quando un dispositivo desidera acquisire il controllo del bus, porta *hold* a 1. Ad esso la CPU risponde con *holda*. Il processore,

terminato il corrente ciclo di bus, pone in alta impedenza i segnali di ABUS e di controllo e riporta a 0 quello di *hold*.

- L1/L2 cache control: usati per verificare che quello scritto nella cache sia coerente con quello scritto in memoria.
- Segnali di Interrupt: essi sono di tre tipologie e sono
 1. INTR: richiesta di Interrupt da un dispositivo esterno
 2. INTA: accettazione della richiesta da parte della CPU
 3. NMI: richiesta di Interrupt non mascherabile

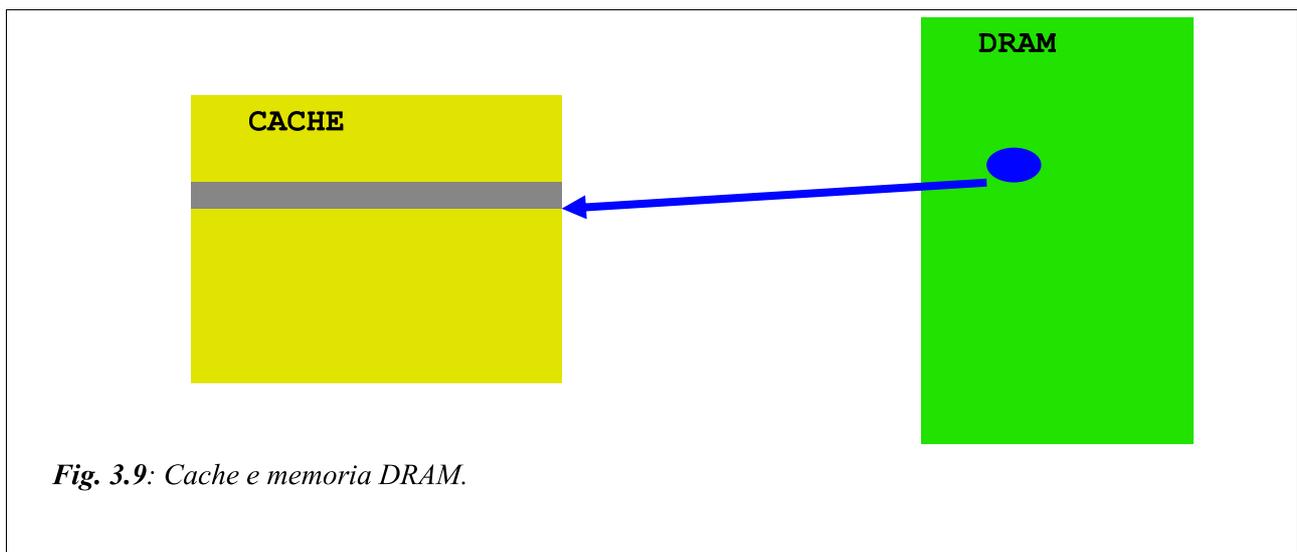
Oltre ai segnali precedentemente enunciati se ne può aggiungere un altro che è quello di LOCK. Esso sta ad indicare che il bus è già usato da un'altra istruzione e quindi non può essere utilizzato da altri. Se premetto ad un'istruzione il prefisso LOCK allora costringo la CPU a eseguirla senza dividere l'esecuzione in più parti.

Fondamentalmente nel Pentium vi sono due tipologie di cicli di bus che sono:

- single transfer: trasferimento di un solo dato (tutti i processori sequenziali sono fatti in questo modo). Nell'utilizzo di DRAM nel caso di single transfer si deve avere che il tempo di accesso alla memoria deve essere minore o uguale al tempo di CPU. Ma, poichè, il tempo di accesso è maggiore del tempo di CPU si inseriscono dei cicli di wait in modo da rispettare questa condizione. Al crescere della frequenza di clock del bus cresce anche il numero di questi cicli di wait.
- Burst cycle: trasferimento di 32 byte (4*64bit) effettuato per l'aggiornamento della L1 cache nei casi di cache miss.

Proprio riguardo a quest'ultima tipologia di ciclo è bene andare a fare alcune considerazioni. Staticamente un processore legge la memoria sostanzialmente per operazioni che riguardano la

cache (aggiornamento); per questo tipo di operazioni valgono i principi di località. Nelle operazioni che interessano la cache si ha adiacenza di indirizzi e un tasso di hit di circa il 98% (se la cache lavora bene). Senza di essa la lettura di dati in memoria avverrebbe in modo sparso.

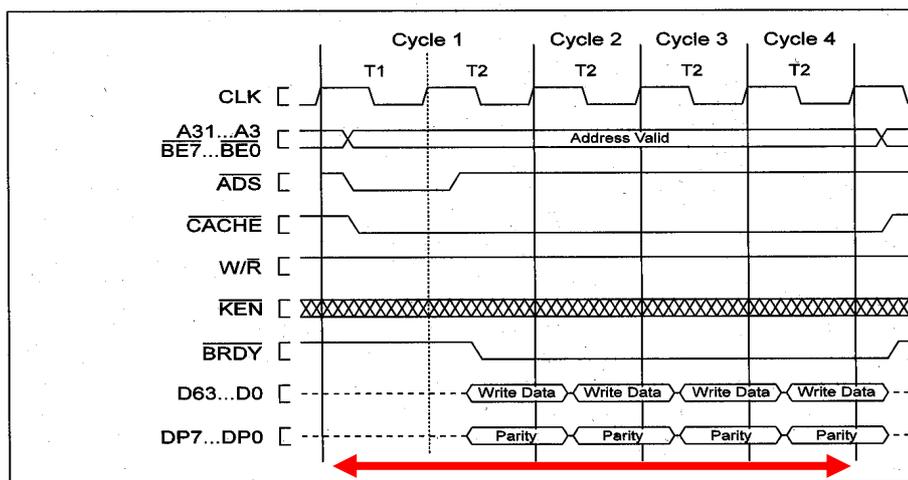


La main memory è utilizzata soprattutto per copiare dati in cache, quindi si cerca di sfruttare l'adiacenza delle celle e inoltre la predizione di indirizzo (in questo modo si diminuisce notevolmente il tempo di accesso). Nei processori odierni con cache on board, le interazioni tra processore e memoria avvengono principalmente per l'aggiornamento della cache in caso di miss oppure per scrittura di questa in memoria. Il parallelismo dei processori dipende dalla gestione efficiente della cache e durante il suo aggiornamento si deve:

- massimizzare il numero di byte associati ad una *cache line* per rendere più efficace il principio di località

- minimizzare il tempo di aggiornamento della cache (numero di cicli di bus per effettuare la lettura della memoria) per minimizzare il tempo di inattività della CPU.

Questi due parametri sono divergenti in quanto migliorandone uno si peggiora l'altro e viceversa. Il compromesso è stato ottenuto utilizzando 4 cicli di bus per aggiornare la cache ed aumentando il numero di segnali del DBUS in modo da inviare più byte per ogni ciclo di bus. In oltre per ottimizzare i tempi di trasferimento si sfrutta il fatto che i byte sono adiacenti e quindi gli indirizzi noti a priori. In tal modo la BIU realizza un ciclo di burst, composto da 4 cicli di bus di cui solo il primo deve essere di 2 clock. I restanti sono di un clock solo. Si ha cioè un ciclo di burst detto "2-1-1-1".



Burst cycle (2-1-1-1)

Figura 3.10: Pentium Burst Cycle.

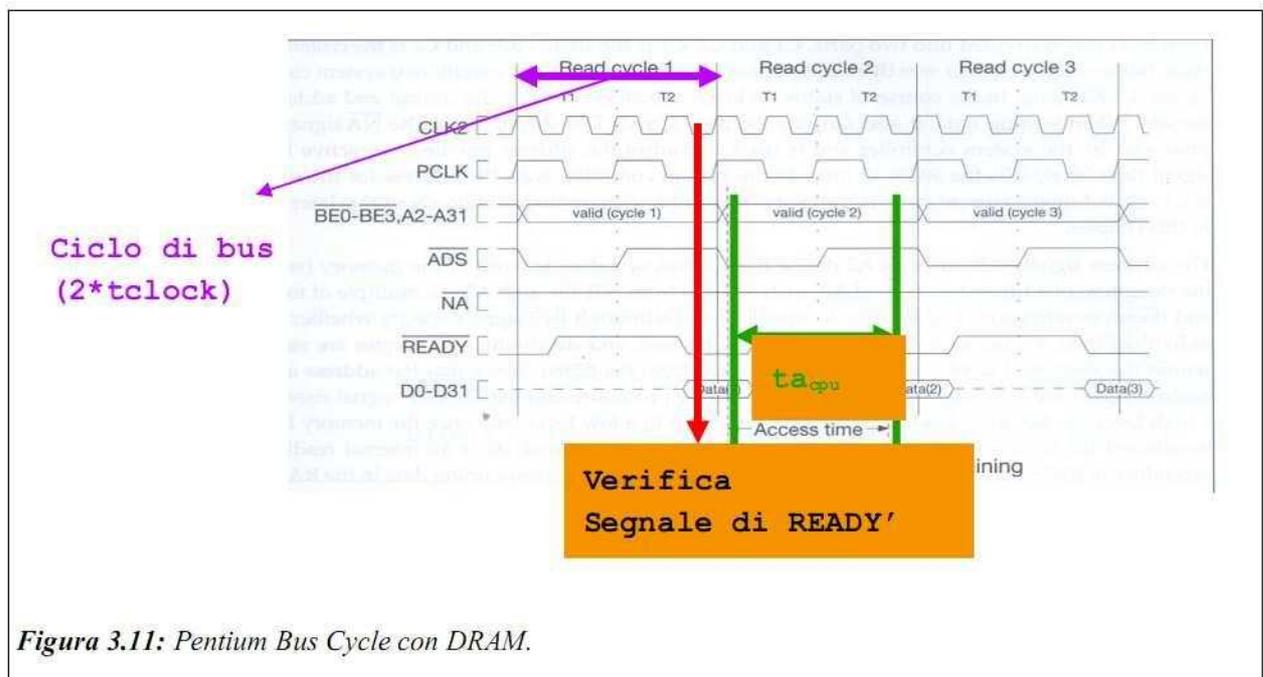
Ad esempio nel caso di bus da 100 MHz, si ha un ciclo burst della durata di $20+10+10+10 = 50$ ns, rispetto agli 80 ns di 4 cicli di bus normali. Va da se che le memorie devono essere in grado di rispondere in tali tempi: da qui le nuove categorie di memoria dette "fast operative", tipo le DDR DRAM.

Infine deve essere fatta un'ultima precisazione riguardo ai bus di sistema che rispetto ai bus di cpu (host bus, FSB) sono oggi prevalentemente orientati alla gestione di interfacce di I/O oltre che alla memoria e non fanno riferimento ad una cache. In aggiunta ai tradizionali segnali di bus di cpu (ABUS, DBUS, CBUS, cache control), i bus di sistema includono almeno i seguenti segnali/funzioni:

- Segnali per interrupt
- Segnali per DMA
- Segnali per arbitraggio bus in ambiente multi master
- Alimentazione

I dispositivi connessi al bus possono essere disposti sulla piastra (motherboard) o connessi tramite connettore (standard).

3.4 Esempio di Pentium Bus Cycle con DRAM



3. Bus di sistema

Si supponga di avere un pentium con host bus a 30MHz che interfaccia una memoria dram con $t_a = 80$ ns. Nel caso di single transfert (trasferimento di un solo dato) sono richiesti un certo numero di cicli di wait. Quindi possiamo dire che essendo l'host bus a 30MHz significa che un ciclo di bus dura:

$$33 \cdot 2 = 66 \text{ ns (tclock} = 33\text{ns)}$$

La cpu acquisisce i dati dal bus (memoria) in circa $\frac{3}{4}$ di un ciclo di bus (come possiamo vedere in fig. 3.11 considerando la larghezza di $t_{a\text{cpu}}$). Nel caso in esame la memoria dovrebbe pertanto rispondere in circa:

$$50 \text{ ns (75\% di 66ns)}$$

Ogni ciclo di wait inserito ha durata (t_w) pari ad un periodo di clock quindi nel caso in esame 33 ns. In generale come già detto in precedenza deve valere la relazione:

$$t_{a(\text{memoria})} \leq t_{a\text{cpu}} + n \cdot t_w \quad (1)$$

che se applicata al caso in esame, si ottiene:

$$80 \leq 50 + n \cdot 33$$

Da cui si deriva che il numero di cicli di wait da inserire è pari ad $n = 1$.

Mettiamoci ora nel caso in cui si abbia un host bus a 100MHz. Questo significa che un ciclo di bus dura:

$$10 \cdot 2 = 20 \text{ ns (tclock} = 10\text{ns), } t_{a\text{cpu}} = 15\text{ns}$$

Applicando la (1) al caso in esame, si ottiene:

$$80 \leq 15 + n \cdot 10$$

Da cui si deriva che il numero di cicli di wait da inserire è pari ad $n = 7$.

3.5 PCI Bus

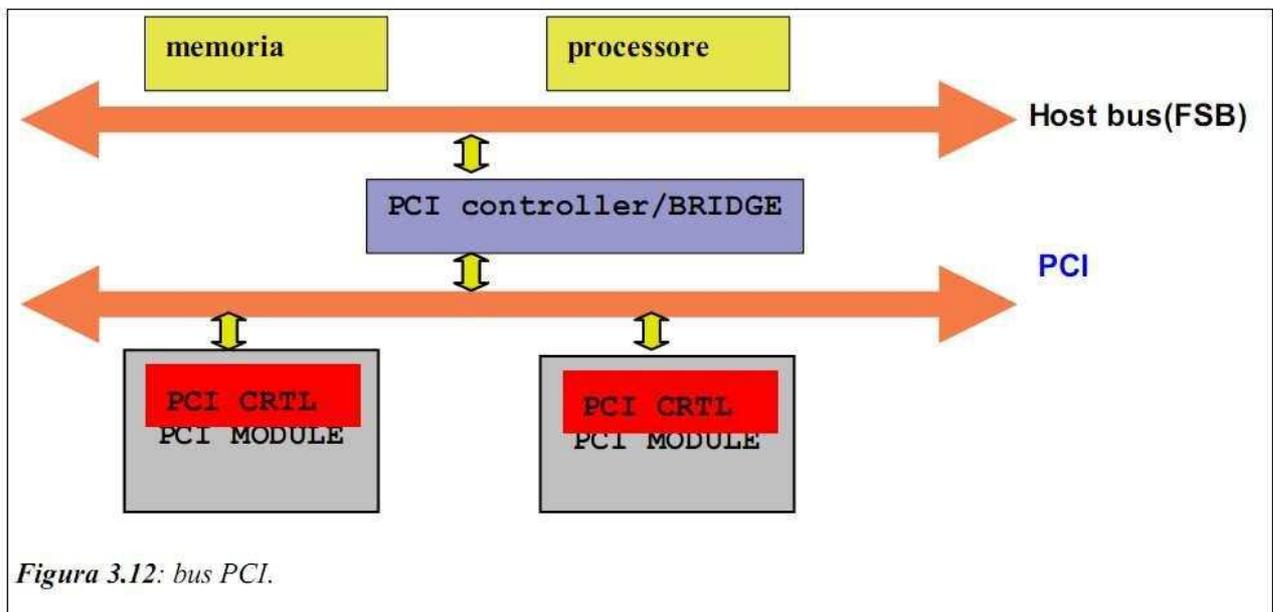


Figura 3.12: bus PCI.

Per cercare di andare a vedere come funzionano i bus di sistema andiamo a considerare quello che sicuramente è il tipo più diffuso: il bus PCI. Nella figura precedente è rappresentata un bus PCI in una tipica architettura di tipo Pentium. Il suo compito è quello di andare a collegare le periferiche esterne con il resto dell'architettura. Ora come è possibile vedere in figura vediamo che sono presenti dei moduli detti PCI controller. Essi hanno il compito di gestire le transazioni sul bus, gestire i registri di configurazione e di agire come *inziator* (busmaster) o *target* (busslave). Ad ogni module è associata un'area di configurazione di 256 Byte, vista come 64 registri da 32 bit, indirizzabili singolarmente mediante cicli di I/O. I primi 4 registri sono standard e contengono informazioni di identificazione del module. Nei PC, nello spazio di I/O di 64K, viene riservato un'area di 4K (c000h – cfffh) per un massimo di 16 moduli: in fase di boot, tramite BIOS, tali aree sono inizializzate.

Altro importante componente in questa tipologia di bus è il PCI-bridge il cui compito è quello di funzionare come un vero e proprio buffer in cui vengono posizionati i dati quando non

immediatamente utilizzabili. Quando, infatti, l'Host bus deve consegnare dei dati al bus PCI che è, in quel momento, impegnato, li deposita all'interno di questo buffer. Appena possibile il bus PCI andrà a recuperare tali dati. Lo stesso meccanismo avviene anche nel verso opposto. È importante ricordare che il bus PCI preleva oltre al dato necessario anche dei dati ad esso adiacente, che sistema all'interno del buffer per un successivo utilizzo.

Per cercare di capire meglio come funziona, in questo caso, un ciclo di bus andiamo ad analizzare la sua tempistica.

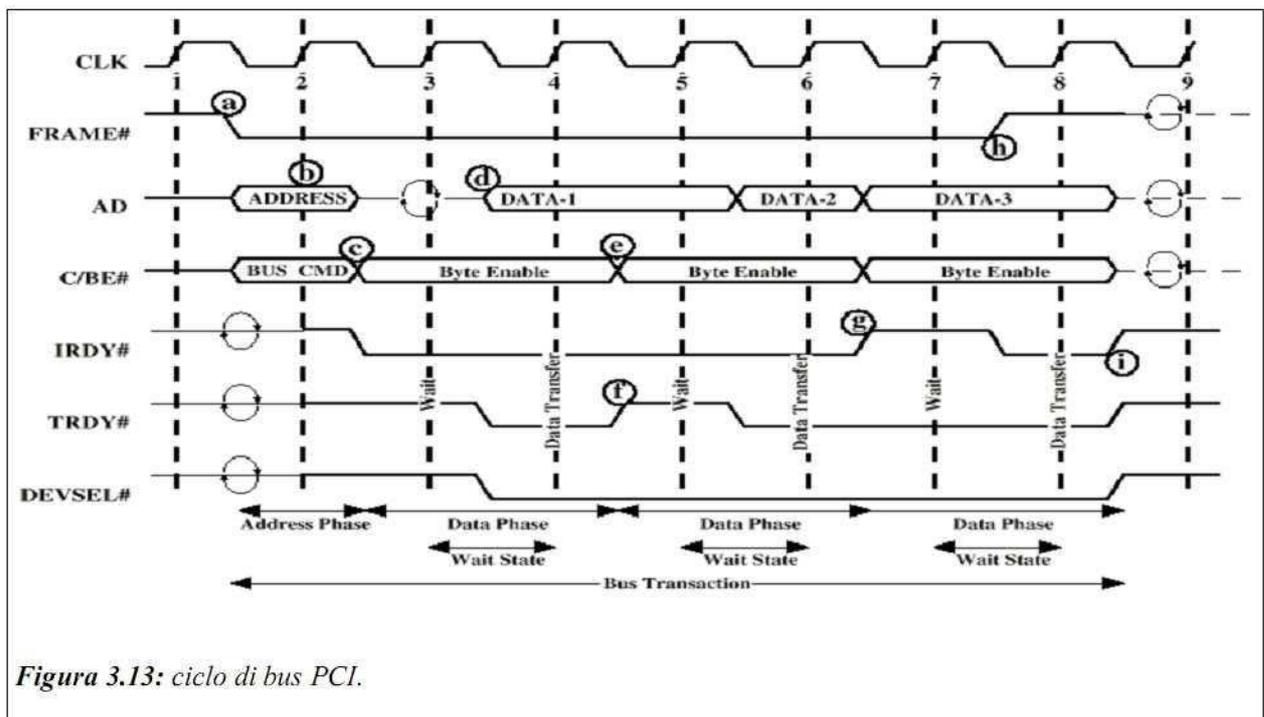


Figura 3.13: ciclo di bus PCI.

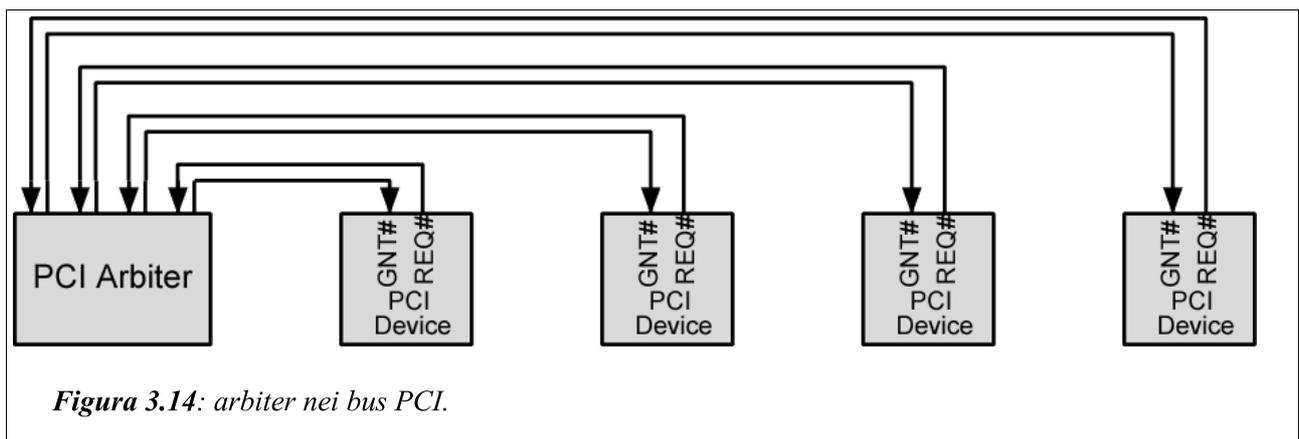
Dopo che all'*initiator* è stato dato, attraverso un arbitraggio, il controllo del bus, viene emesso il segnale di Frame il cui compito è quello di segnalare l'inizio di un ciclo. Vi è quindi (come in un ciclo di bus normale) la gestione degli indirizzi che è seguita da tutta una serie di segnali la cui funzione è quella di definire la tipologia di ciclo. Una parte di questi segnali è:

- Interrupt INTA (0000)

- Special (0001)
- I/O read (0010)
- I/O write(0011)
- Memory read (0110)
- Memory write (0111)
- Configuration read (1010)
- Memory multiple read (1100)

Ad essi segue quello di Ready che può portare, a seconda delle necessità, all'inserimento di cicli di Wait. Da qui in poi, praticamente, si segue la tempistica di un normale ciclo di bus.

Ma a questo punto ci chiediamo: come avviene l'arbitraggio nel bus PCI? Per rispondere a questa domanda facciamo riferimento alla figura seguente:



Il tutto viene gestito attraverso l'utilizzo di segnali dedicati di tipo *request* e *grant*. Perciò il protocollo del PCI non gestisce il tipo di arbitraggio (la cui scelta è completamente lasciata al programmatore) ma bensì va a definire solo quale deve essere il meccanismo di richiesta e di risposta: ogni dispositivo deve essere dotato quindi di questi due segnali e, a seconda del tipo di arbitraggio prescelto, viene assegnato il controllo del bus.

4. Sottosistema di memoria nell'architettura 80x86

In questo capitolo verrà affrontato il discorso relativo alle memorie in un sistema a processore con particolare attenzione in riferimento al sistema 80x86.

Gli argomenti che verranno trattati sono:

- I tipi di DRAM (Data RAM)
- I circuiti di decodifica della memoria
- DRAM controller
- Progetto di un banco di memoria

Quando si parla di memoria, è necessario andare ad analizzare quali sono le problematiche che introduce l'utilizzo della stessa. Le problematiche principali da analizzare sono:

- La grande differenza tra la velocità di risposta della memoria e i tempi di risposta attesi dalla CPU
 1. Soluzioni tecnologiche (fast operative mode DRAM) cioè utilizzare dei dispositivi di memoria (come ad esempio le DDR Double Data Rate), che hanno un architettura interna tale da permette in certe condizioni di reggere alla velocità del bus
 2. Soluzioni architetturali (interleaving)
- L'esigenza (da parte delle memorie di tipo dinamico) di cicli di refresh per non perdere i dati
- Rilevazione e correzione degli errori
- Prestazioni del DRAM controller e chip set

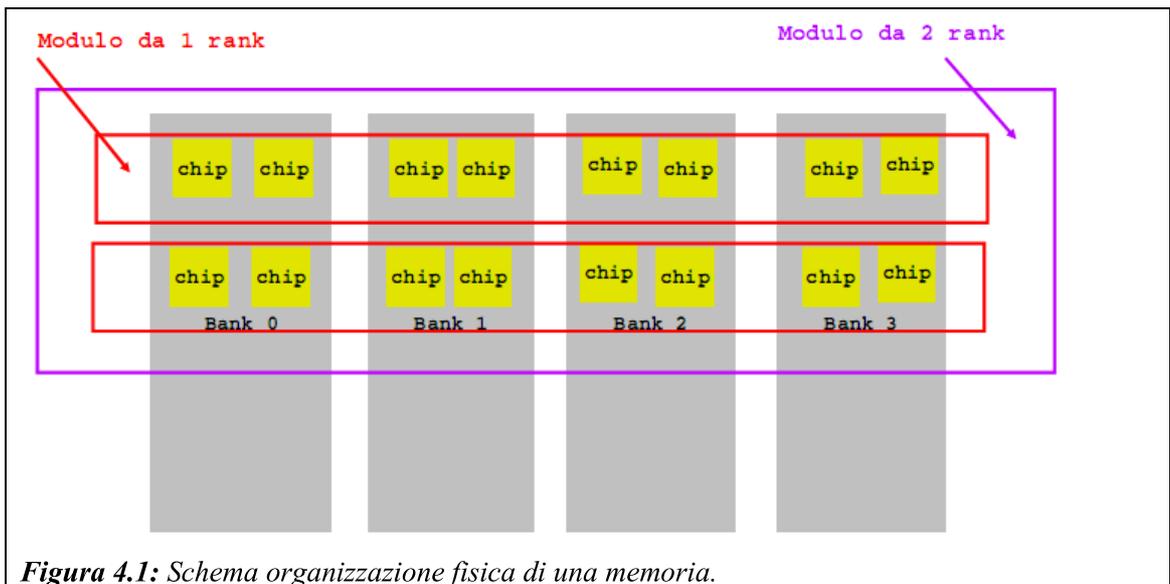
4.1 Organizzazione fisica di una memoria: aspetti terminologici

Un blocco di memoria può essere pensato, per semplicità, come una matrice. In riferimento a questa matrice si definiscono gli aspetti terminologici.

- **Device:** singolo chip caratterizzato da una profondità (numero di indirizzi) e da un parallelismo dati (bit di dato). I chip sui datasheet sono indicati secondo la seguente modalità: $nK \times m$, dove n è la profondità del chip, cioè il numero di indirizzi, m è il numero di bit su cui è articolato e K è la dimensione (K, M, G...).

Ad esempio, per costruire una memoria da 1Gb possiamo utilizzare chip da 1G x 1b, 0.5G x 2b, 0.25G x 4b etc. (si vedrà più avanti un esercizio)

- **Banco (o blocco):** blocco fisico/logico con parallelismo di un byte selezionato dai segnali BE_i (Bank Enable). Il numero dei banchi dipende dal parallelismo del DBUS. In riferimento alla matrice il banco è identificato dalla colonna
- **Rank:** porzione di un modulo costituito da un blocco di device con parallelismo pari al DBUS e profondità pari a quelle dei chip. Il rank è invece identificato dalla riga della matrice
- **Modulo:** insieme di device con parallelismo pari al DBUS (tutti i banchi) realizzato su unica piastrina (SIMM, DIMM,..). Può essere composto da uno o più rank



Esempio:

Si vuole avere una memoria complessiva da 1MB. Si dispone di un ABUS da 32 bit (4byte) e chip da 64K x 4b (256Kbit).

Il DBUS da 32 bit indica che si devono avere 4 banki, poiché su di esso si devono mandare 4 byte per volta. Un rank è composto da 8 chip poiché ogni chip è articolato su 4 bit, quindi $(32/4)=8$;

Ogni rank da 8 chip = 256 KB, quindi per arrivare ad 1MB necessitano 4 rank.

La memoria ha quindi 4 banki, 4 rank per un totale di 32 chip. (si tralascia in questo esempio la struttura del DBUS che verrà ripresa in seguito).

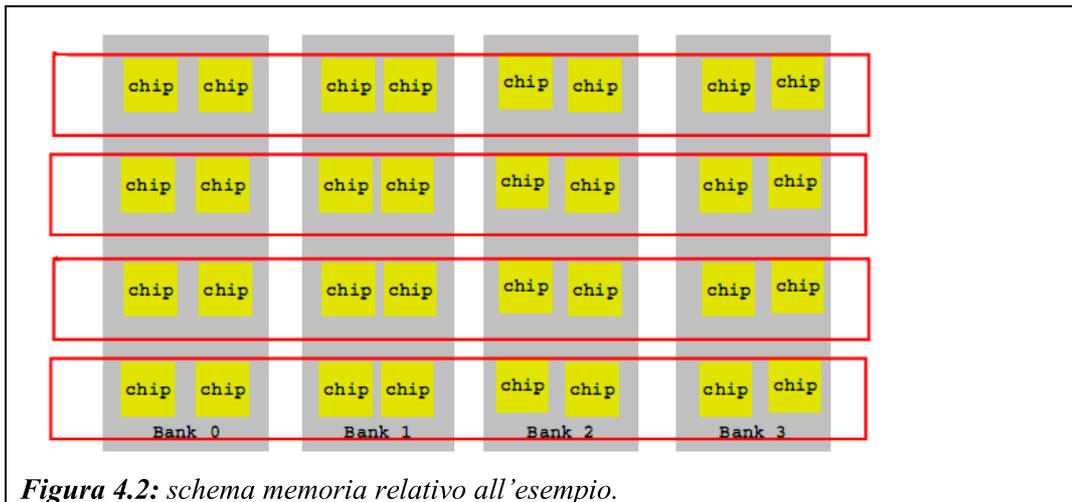


Figura 4.2: schema memoria relativo all'esempio.

4.2 Organizzazione di una memoria

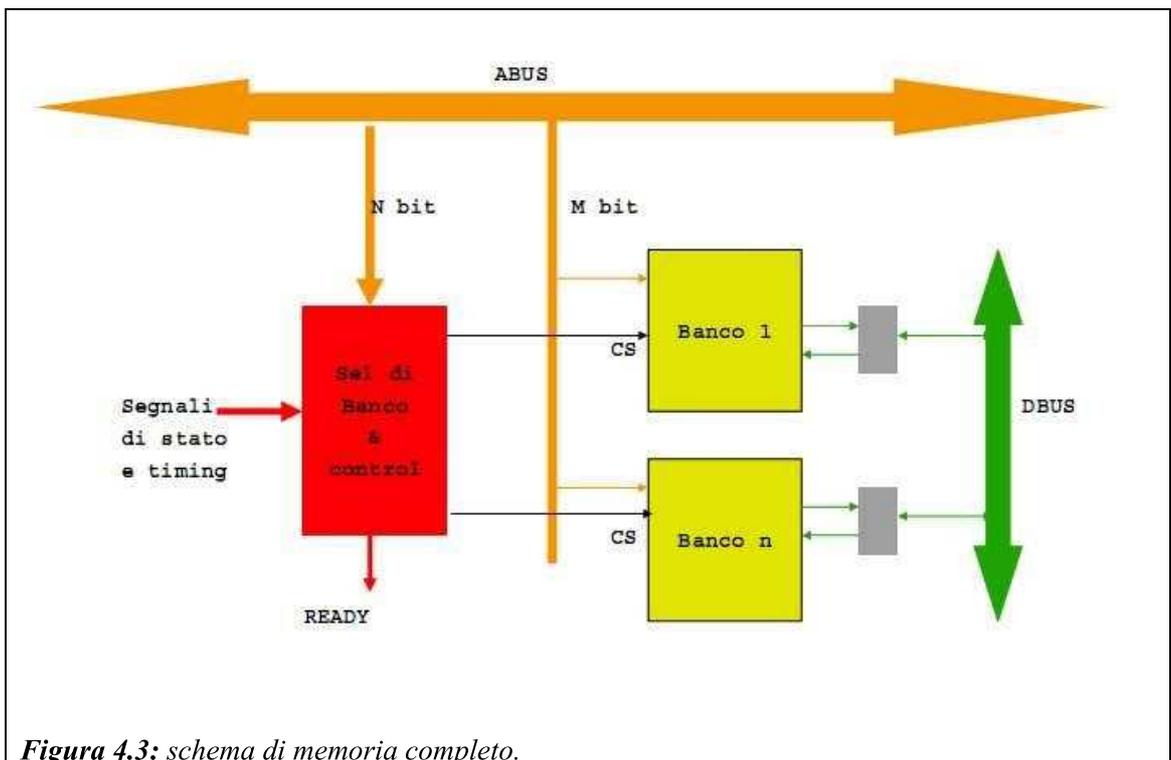
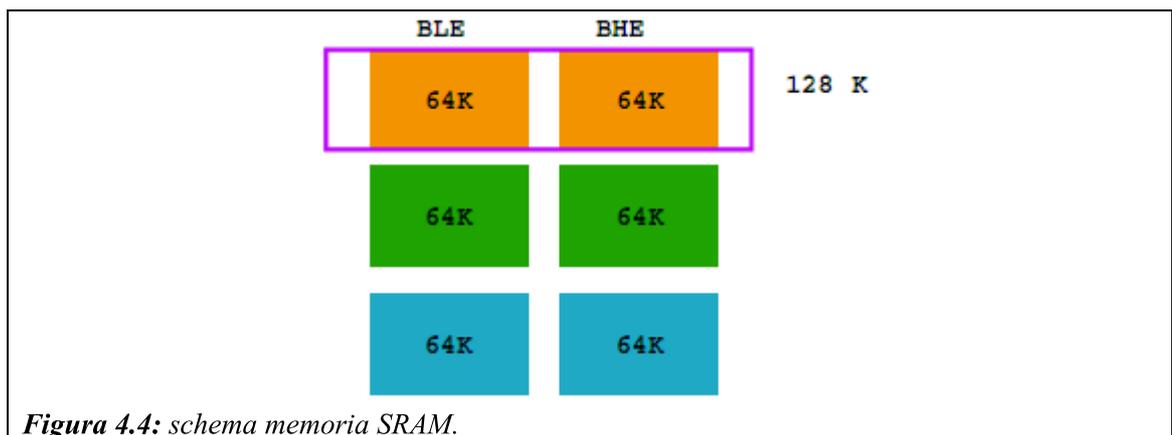


Figura 4.3: schema di memoria completo.

Ogni banco di memoria ha dimensione 1M, quindi si ha una memoria totale di 2M. L'ABUS sarà quindi un BUS da 21 bit ($2^{21} = 2M$), di cui 20 bit selezionano la parola all'interno del singolo banco e un bit seleziona il banco stesso. Il banco 1 conterrà le parole relative agli indirizzi che vanno da 0M a (1M-1), mentre il banco 2 da 1M a 2M. Il circuito di selezione del banco (in questo esempio) è costituito da un solo bit, in particolare il bit 21°, che se uguale a 0 seleziona il banco 1 altrimenti seleziona il banco 2. Il circuito di selezione del banco tiene conto anche del timing e dei segnali di stato, che indicano che la memoria è attiva (ad esempio il segnale M/IO dove M indica un indirizzo relativo alla memoria e IO un indirizzo relativo ad un periferico). Il circuito di selezione del banco ha anche il compito di generare il segnale di ready, poiché conosce il tempo di risposta dei chip e quindi in base a questo è in grado di calcolare il tempo che intercorre dalla richiesta a quando i dati sono pronti sul BUS.

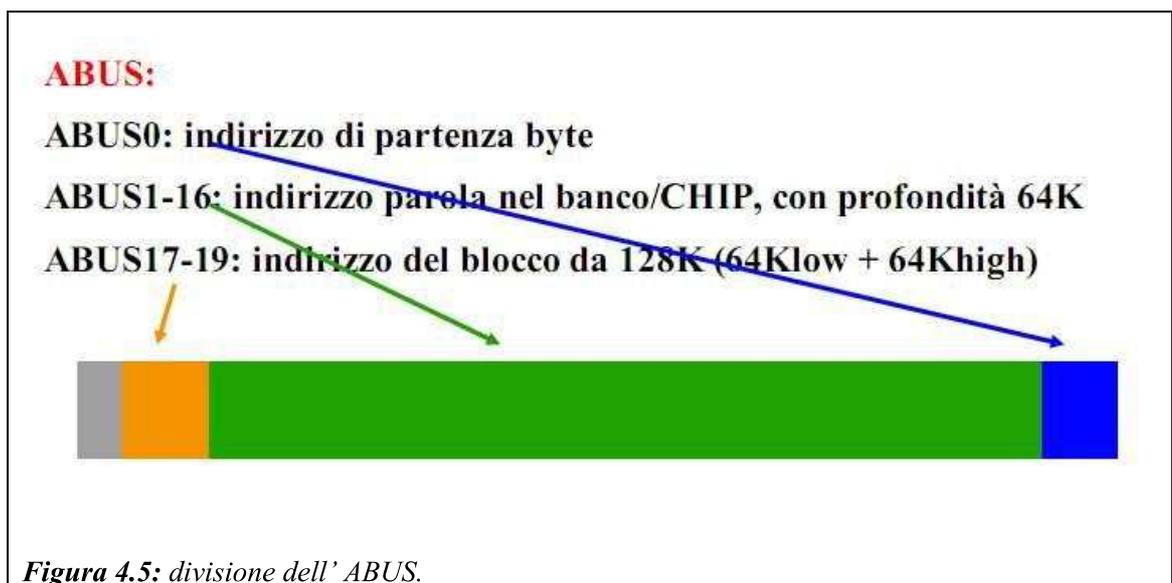
4.3 Banco di memoria SRAM



Nell'architettura 80x86, la memoria è organizzata in due banche, quindi si hanno due segnali di BE (bank enable), chiamati BHE (Bank High Enable) e BLE (Bank Low Enable). Per capire meglio come è organizzato l'address BUS, si consideri il seguente esempio:

Si supponga di avere una memoria da 1MB con un DBUS da 16 bit costituita da 8 chip da 64Kx8b.

Poiché i chip hanno profondità 64K, sono necessari 16 bit per individuare la parola all'interno dello stesso chip e 1 bit per selezionare il banco in cui è contenuta la parola. In generale un ABUS sarà un BUS suddiviso come segue (ipotesi di ABUS a 20 bit):



La memoria ha quindi una struttura circuitale di questo tipo:

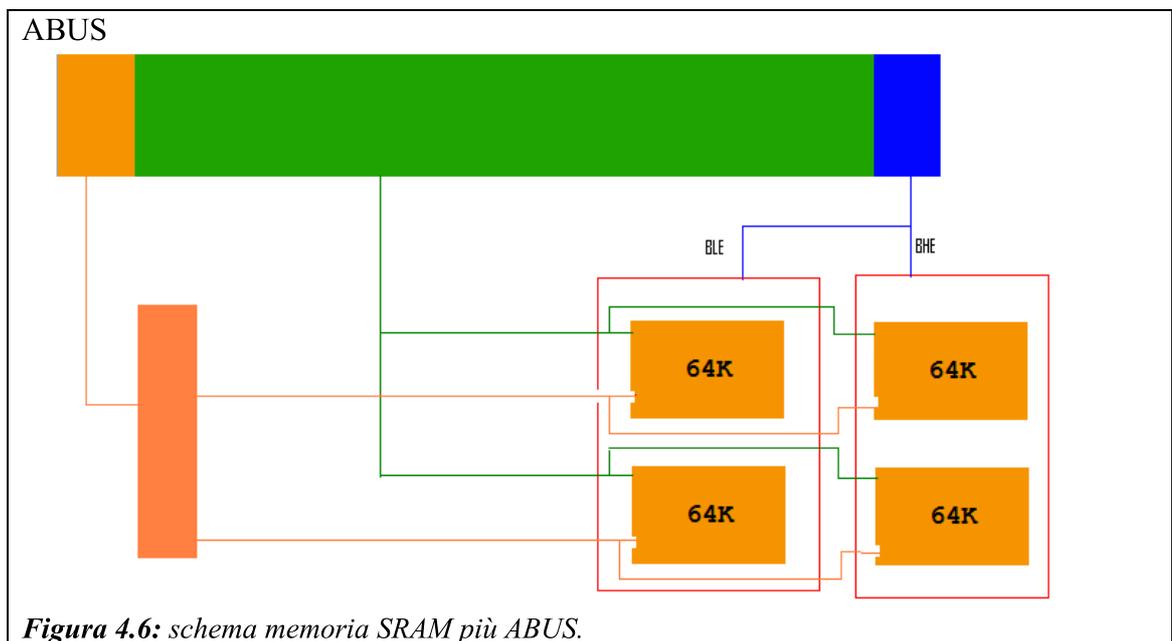


Figura 4.6: schema memoria SRAM più ABUS.

Il bit 'blu' è il bit che seleziona il banco di interesse (il bit di BE), i bit in 'verde' sono quelli che selezionano la parola all'interno del chip (vengono mandati a tutti i chip) e i bit in 'arancio' vengono mandati in ingresso ad un'opportuna circuiteria che seleziona il chip select.

Se si ha un'istruzione del tipo (ipotesti ABUS = 20 bit):

MOV AX, [20002H] dove 20002H = 0010000000000000010

verrà selezionato il banco di sinistra (BLE), la parola all'interno del chip si trova all'indirizzo 1 (quindi è la seconda parola) e il rank è il rank numero 1. N.B. Il segnale di BE indica solo il banco da cui iniziare a leggere, l'istruzione indica invece quanti byte leggere. In questo esempio si vede chiaramente che si inizia a leggere dal banco meno significativo, ma il registro AX (16 bit) implica la lettura di 2 byte.

Di seguito uno schema di un banco di SRAM completo:

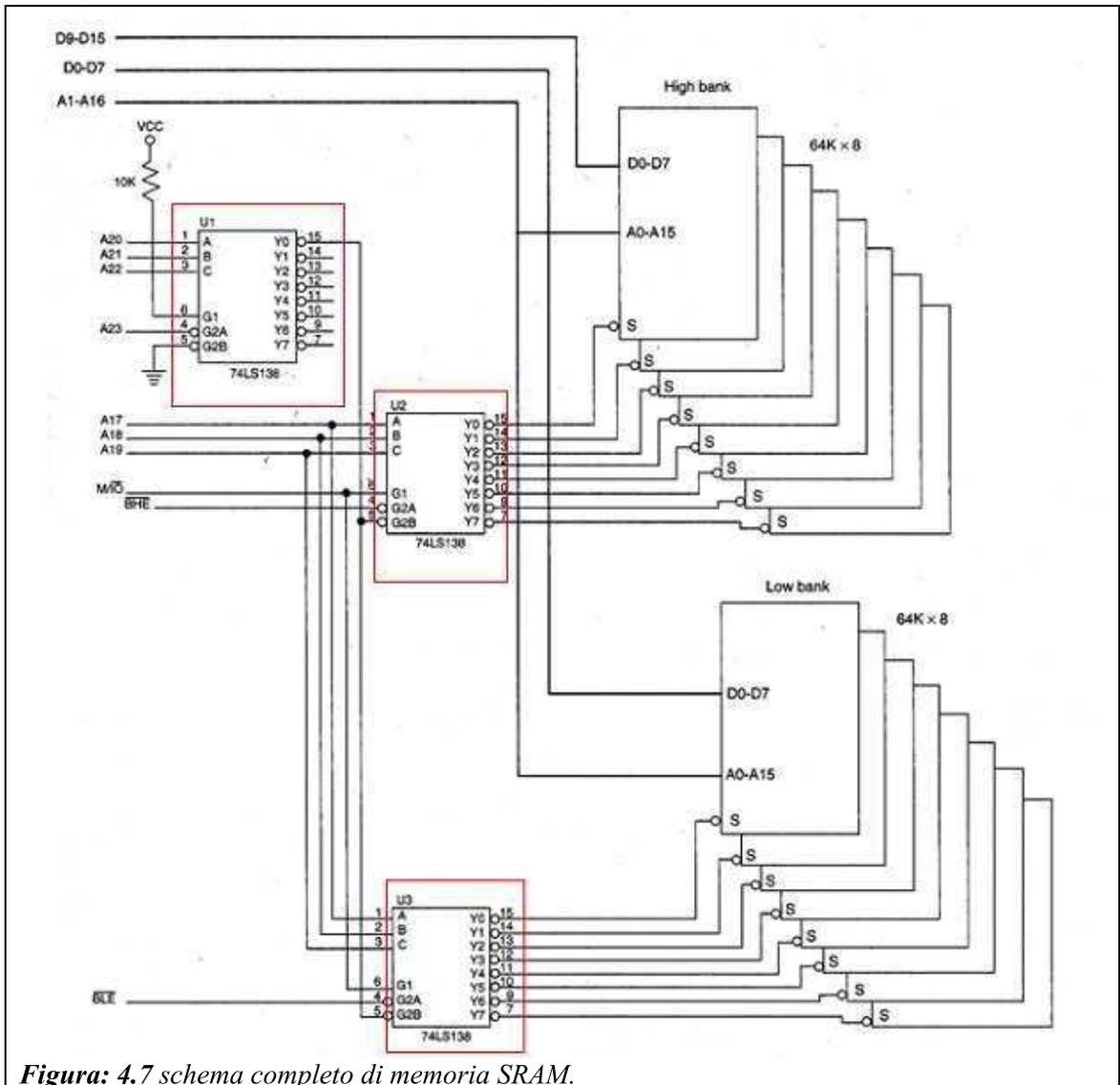


Figura: 4.7 schema completo di memoria SRAM.

I blocchi selezionati in rosso sono dei decoder (n ingressi 2^n uscite) che vengono abilitati dai segnali di controllo gestiti direttamente dal processore. Questi segnali di controllo vengono ricavati dal codice operativo dell'istruzione e servono per attivare i decoder quando sul ABUS c'è un indirizzo relativo alla lettura o alla scrittura di un dato da e su memoria.

4.4 Breve cenno sull'interleaving

Si ricorda che ogni memoria ha la sua tempistica. In particolare si ha:

- Tempo di accesso: tempo che intercorre da quando si inizia l'operazione a quando si accede al dato relativo (circa 40 ns)
- Tempo di ciclo: tempo che intercorre dall'inizio alla fine dell'operazione, ossia fino a quando non si ha la possibilità di iniziare un'altra operazione (circa 100 ns)

Nonostante la risposta arrivi in 40 ns, per 60 ns non si possono leggere dati da quel chip, questo punto è un punto critico che costringe ad inserire cicli di wait.

La tecnica dell'interleaving permette di ridurre questa criticità. La memoria è organizzata in modo che tutti i dati relativi ad indirizzi pari stiano su un chip (o meglio su un banco se composto da più chip) e tutti i dati relativi ad indirizzi dispari su un altro chip (o meglio un altro banco). La probabilità di dover leggere per due volte consecutive dati relativi ad indirizzi pari (o ad indirizzi dispari) è molto bassa. Poiché per leggere un dato da un indirizzo pari servono 40 ns, non sarà più necessario aspettare 60 ns ma si andrà a leggere in un altro chip dove si trovano i dati relativi agli indirizzi dispari. Se si deve leggere consecutivamente da indirizzi pari (o dispari) bisogna per forza attendere 60 ns. Questa tecnica necessita di una circuiteria molto complessa in grado di distinguere gli indirizzi pari da quelli dispari e di inserire i cicli di wait opportuni (diversi per letture consecutive su indirizzi pari o indirizzi dispari da letture su indirizzi 'alternati'). Questa tecnica non è spesso utilizzata perché si preferisce usare le DDR, talvolta le due soluzioni vengono utilizzate contemporaneamente per ottenere prestazioni maggiori.

4.5 Le memorie DRAM

Per ragioni di semplicità espositiva si fa riferimento ad un chip da 1Mb, rispetto agli attuali da 1Gb.

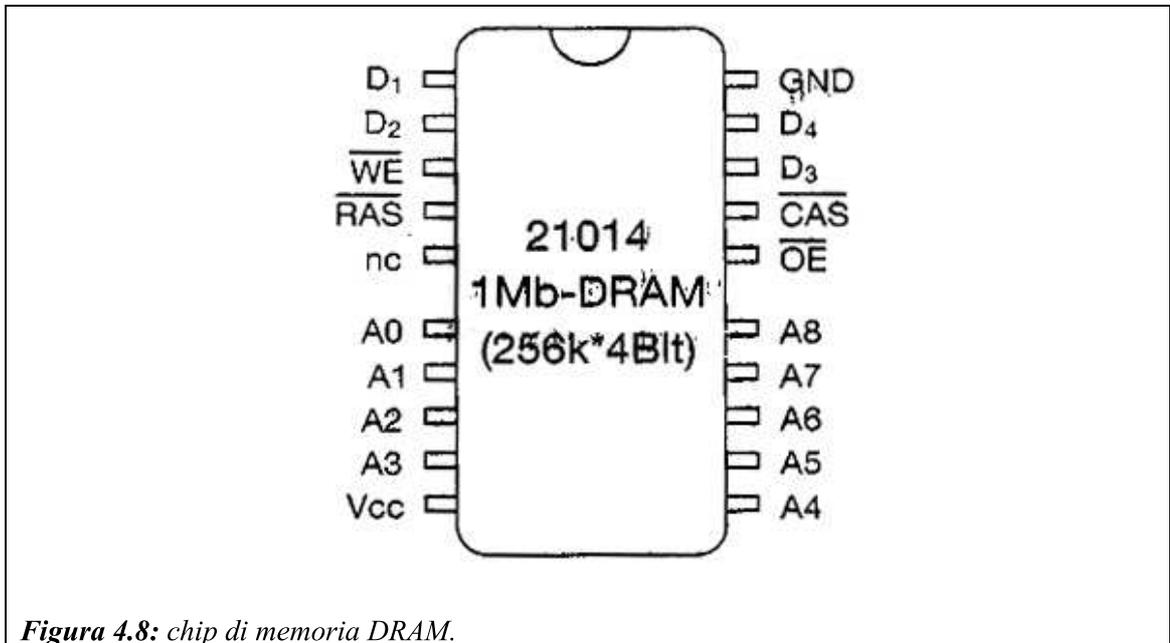


Figura 4.8: chip di memoria DRAM.

Le memorie di tipo dinamico, al contrario di quelle statiche, hanno la caratteristica di avere il BUS degli indirizzi moltiplicato dai segnali di RAS e CAS, al fine di ridurre il numero di piedini.

Nelle memorie di tipo dinamico, infatti, viene mandato sul BUS l'indirizzo diviso in due parti (non necessariamente uguali perché la matrice di memoria non è per forza quadrata), ossia prima viene mandata la prima parte dell'indirizzo (chiamata indirizzo di riga) e poi la seconda parte (chiamata indirizzo di colonna).

La moltiplicazione necessita della sincronizzazione delle due parti dell'indirizzo, questa viene fatta attraverso l'utilizzo dei segnali chiamati:

- RAS: Row Address Strobe
- CAS: Column Address Strobe

Anche per le memorie di tipo dinamico, si può pensare lo schema come una matrice (per semplicità quadrata, ma non è sempre così). Il numero di piedini esterni è dimensionato in base al numero massimo che serve per l'indirizzo di riga o di colonna.

Esempio:

siano:

- $\lg_2 m = 10$;indirizzo di riga
- $\lg_2 n = 9$;indirizzo di colonna

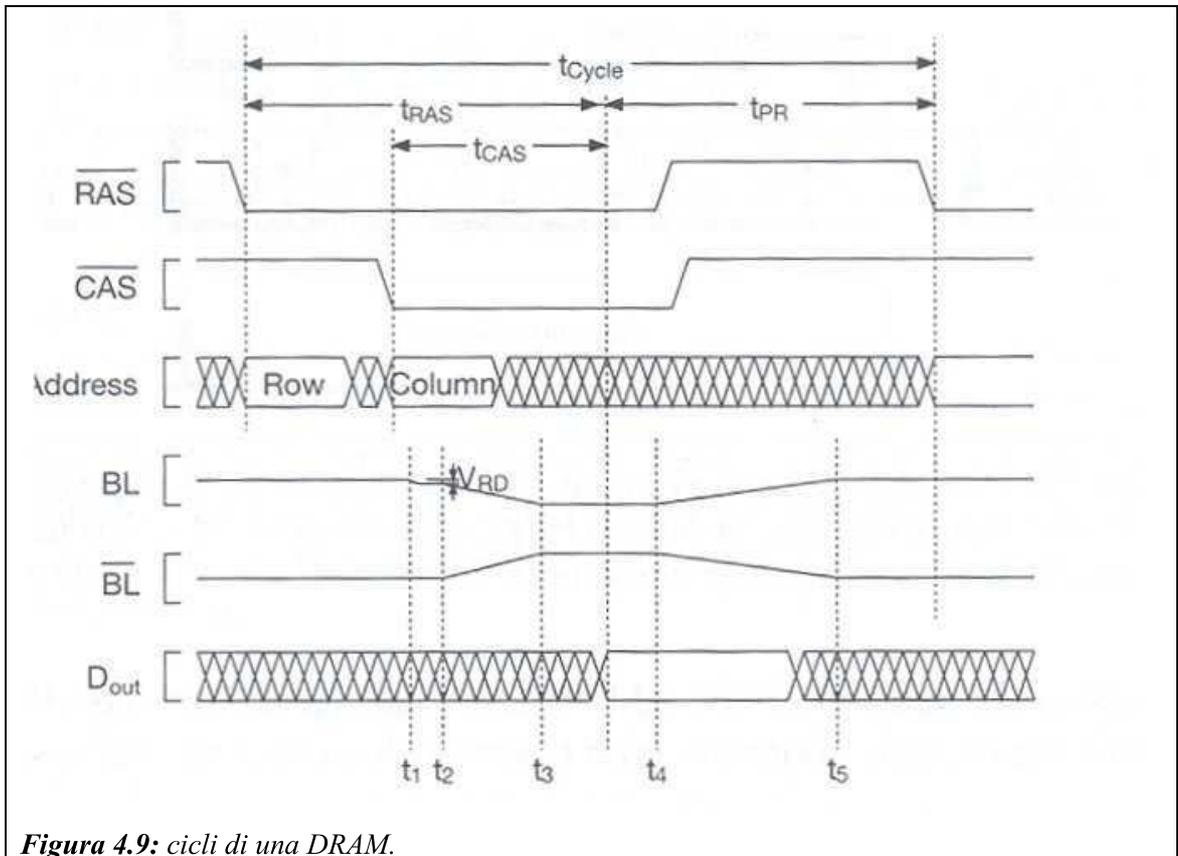
Si avranno un numero di pin esterni pari a 10, perché solo 9 non basterebbero per trasmettere l'indirizzo di riga.

In generale si può dire che, se C è la capacità della memoria, $\lg_2 C$ sono il numero di bit (e quindi di piedini) di indirizzo. Con il metodo multiplato invece il numero di bit (e quindi di piedini) di indirizzo sono pari a $\frac{\lg_2 C}{2}$.

I cicli che si possono distinguere in una DRAM sono:

- ciclo di read
- ciclo di write (2 tipi)
- ciclo di refresh
- ciclo fast operative (cicli che permettono di accedere più velocemente a dati adiacenti).

4.5.1 Ciclo di base di una DRAM



I segnali di RAS e CAS sono attivi bassi, quando RAS = 0 sull'ABUS si manda l'indirizzo di riga, quando diventa basso anche il segnale di CAS (si noti che il segnale RAS continua a rimanere basso), si manda l'indirizzo di colonna. Dopo un certo intervallo di tempo il dato è disponibile sui piedini di uscita per la lettura altrimenti il dato si mette sul DBUS per la scrittura. Si prende come esempio un ciclo di lettura, dal punto di vista della temporizzazione si ha:

- t_{ras} : (uguale tempo di accesso) tempo che intercorre tra il fronte di discesa del segnale RAS e la disponibilità dei dati sui piedini di uscita
- t_{cas} : tempo che intercorre tra il fronte di discesa del segnale CAS e la disponibilità dei dati sui piedini di uscita

- t_{cycle} : tempo totale, che tiene conto del fatto che i segnale di RAS e CAS devono tornare alti per un lasso di tempo (t_{pr}) altrimenti non sarebbe possibile distinguere la fine di un ciclo dall'inizio di un altro ciclo

In definitiva possiamo dire che il tempo t_{ras} è il tempo necessario per leggere il dato, mentre il tempo $t_{pr} = t_{cycle} - t_{ras}$ è un tempo di inattività, cioè un tempo durante il quale il dispositivo non può essere utilizzato perché non risponde. Dal punto di vista reale è bene sapere che il tempo t_{cycle} è circa pari al doppio del tempo t_{ras} .

4.5.2 Ciclo di refresh

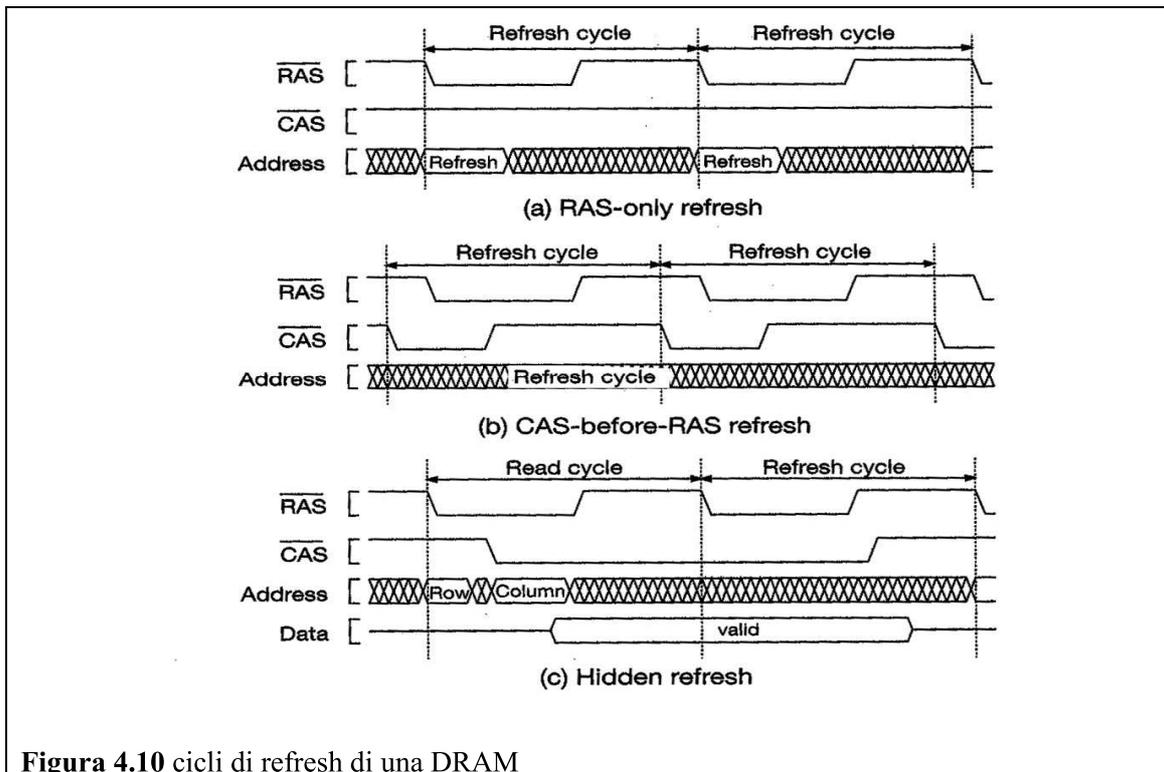


Figura 4.10 cicli di refresh di una DRAM

Il ciclo di refresh è il ciclo che serve ad una memoria dinamica per far sì che il dato in essa contenuto non venga perso.

Ne esistono due tipi:

- Con tempistica interna
- Con tempistica esterna

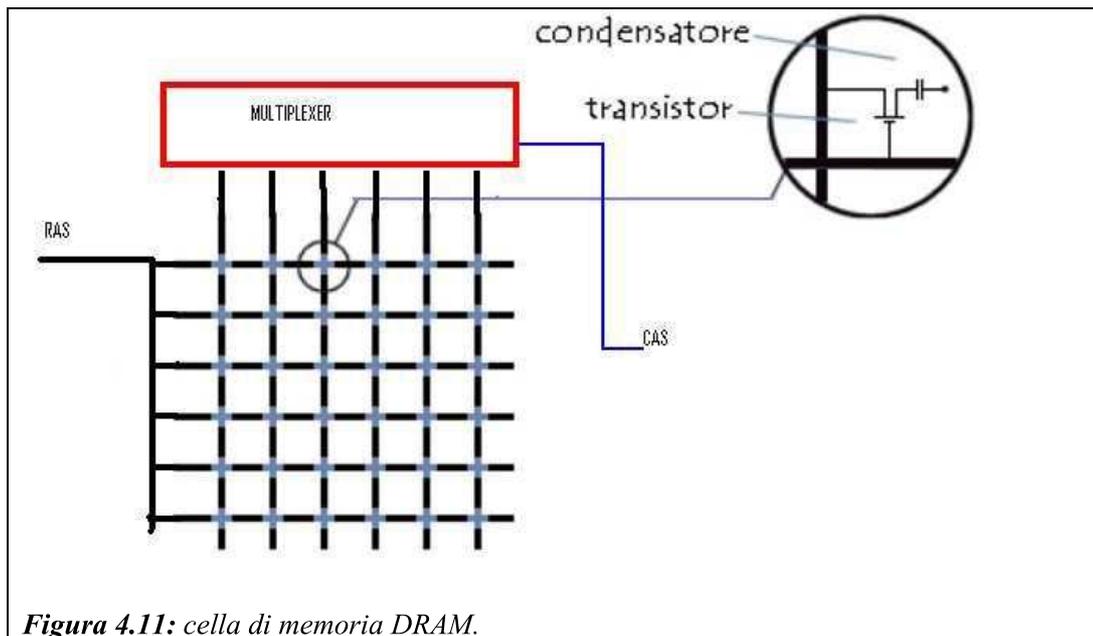
Il ciclo di refresh è sostanzialmente un ciclo di lettura interna di una intera riga. Questa lettura consente di ripristinare la carica all'interno del condensatore, poiché il dato viene letto, viene fatto passare all'interno di un amplificatore e viene infine rimandato dentro la cella sufficientemente amplificato.

4.5.3 Fast Operative Mode

Queste memorie si basano sulla possibilità tecnologica di selezionare celle adiacenti della matrice senza dover completare un ciclo di RAS. Sostanzialmente con questo tipo di memoria si possono leggere tutte le celle associate ad una riga. Vengono adoperate quando si devono fare trasferimenti di dati con indirizzi adiacenti, come nel caso dei cicli burst per aggiornare la cache. In tal caso si leggono tanti byte adiacenti quanti contenuti in una linea di cache. Esistono tre tipi di Fast operative mode:

- Asincrono
- Sincrono
- Protocol based

La memoria è organizzata come mostrato in figura 4.11:



Dove le linee sono fili conduttori e la singola cella è composta da un transistor (switch) e da un condensatore. Le linee (“righe” e “colonne”) non si intersecano tra loro. Quando sulla riga viene mandato il valore logico 1 (cioè viene fatta passare una tensione), il transistor diventa un corto che fa sì che la carica contenuta all’interno del condensatore venga portata sul filo di uscita (la “colonna”). Il valore di RAS abilita un’intera riga, mentre il valore di CAS, che entra in un multiplexer, seleziona la “colonna” in uscita. Il segnale RAS manda in corto tutti i transistor di un’intera riga, portando sui fili di uscita tutti i dati inerenti quella riga. Lo scopo delle fast operative mode è quello di riuscire a commutare velocemente tra una “colonna” e l’altra senza aspettare il successivo ciclo, cioè utilizzando lo stesso segnale di RAS. Così facendo si riesce a leggere velocemente dati adiacenti.

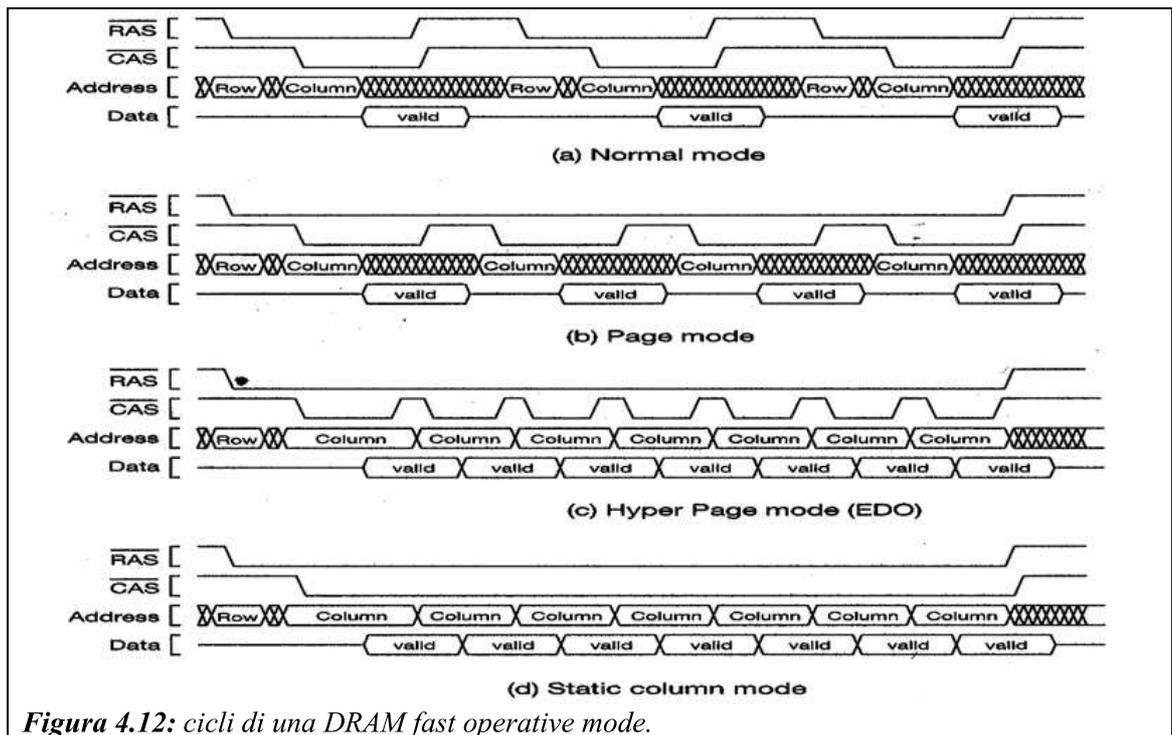


Figura 4.12: cicli di una DRAM fast operative mode.

Il ciclo (a) è il ciclo normale di una RAM dinamica. I cicli (b), (c) e (d), tengono costante il segnale di RAS, si ottengono quindi dati generando solamente indirizzi di colonna. Nei cicli di tipo (b) e (c), si utilizza il segnale di CAS come segnale di sincronizzazione, mentre nel ciclo di tipo (d) il segnale di sincronizzazione si trova all'interno del chip stesso, quindi si possono mandare indirizzi di colonna senza avere mai un fronte di salita, poiché non c'è necessità di far capire quando si cambia l'indirizzo, il chip lo sa già di suo.

4.5.4 La famiglia delle DRAM

Le DRAM si dividono in due grossi "rami" che si dividono a loro volta:

- DRAM asincrone

1. EDO RAM

2. BEDO RAM

- DRAM sincrone

1. SD RAM

2. DDR RAM

Esiste inoltre la versione DRAM Protocol Based, introdotta dalla Intel per i propri processori. Questa versione aveva prestazioni leggermente superiori alle prime DDR, ma è stata soppiantata dalle DDR3 che hanno maggiori prestazioni.

In generale si ha che le RAM di tipo sincrone hanno maggiori prestazioni, poiché non hanno bisogno di scambiare segnali per la sincronizzazione (operazione che richiede del tempo).

4.5.5 Temporizzazione delle SDRAM

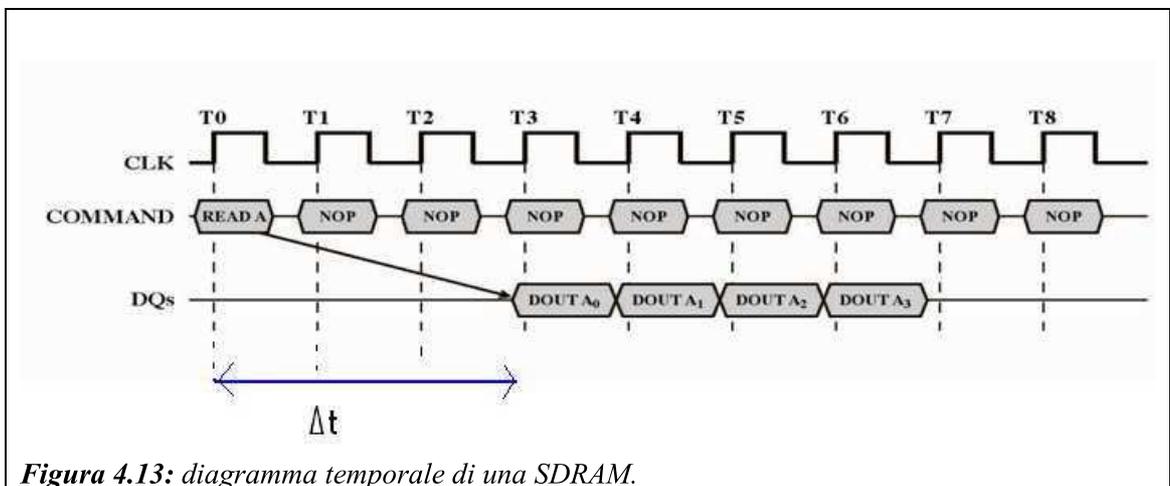


Figura 4.13: diagramma temporale di una SDRAM.

Il ciclo inizia con il fronte di salita di T0, sostanzialmente in questo istante viene attivato il segnale di RAS, questa operazione è fatta in maniera sincrona con il clock del BUS. Dopo un certo intervallo di tempo Δt (che corrisponde al tempo di accesso) i dati sono pronti. Nei clock successivi

i dati vengono inviati con le modalità prima descritte di lettura di un'intera riga.

La velocità data dai progettisti è sempre la velocità di picco, cioè la velocità che si ha a partire dal primo dato pronto (non viene considerato Δt).

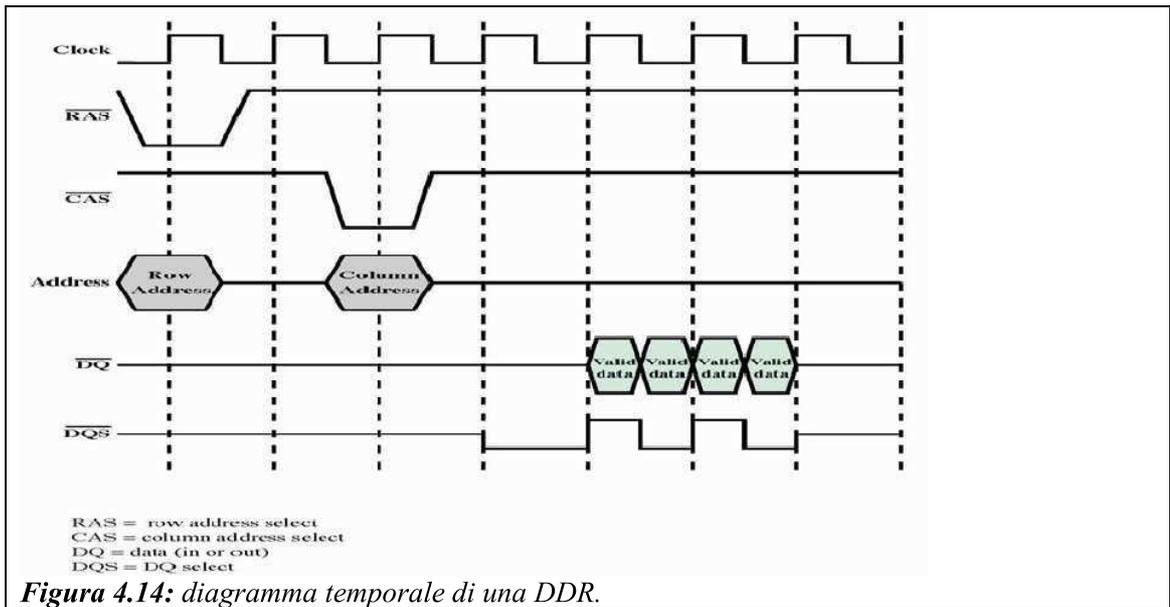
Esempio:

BUS a 100MHz con parallelismo 32 bit = 4 byte $\rightarrow V_{max} = 400 \frac{\text{Mbyte}}{\text{s}}$ senza tener conto del tempo di accesso. A questo punto per completare le informazioni si ha bisogno di sapere il numero di colpi di clock di latenza seguiti dal numero di colpi di clock richiesti per il trasferimento di ciascun dato, si supponga che questa informazione sia 5/1/1/1.

A questo punto si può sapere anche quanto è il tempo di accesso, una frequenza di 100MHz \rightarrow 10 ns, quindi prima che il dato sia pronto passano $5 \cdot 10 \text{ ns} = 50 \text{ ns}$, dopo questo tempo iniziale si ottiene un dato ogni 10 ns.

Le DRAM odierne hanno la possibilità di essere programmate, nel senso che danno la possibilità di definire le caratteristiche del timing, ad esempio può essere programmata la lunghezza del burst.

Il passaggio dalle SDRAM alle DDR deriva dall'utilizzo di un moltiplicatore di frequenza che permette di prelevare dati sia sul fronte di salita che su quello di discesa del clock, non a caso si chiamano Double Data Rate.



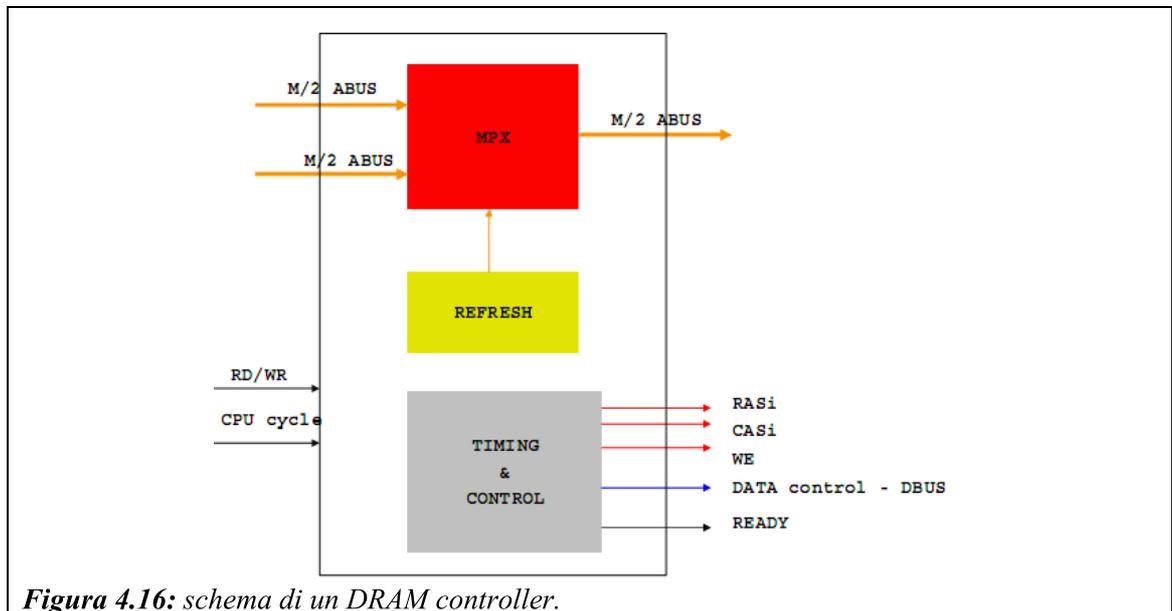
Il raddoppio della frequenza viene fatto internamente alla memoria, sarebbe impossibile farlo esternamente. Il problema della latenza non viene risolto, ma la moltiplicazione di frequenza permette l'invio di più dati.

Alcuni dati relativi alle DDR:

Chip TYPE	Memory Clock	Bus clock	Cycle time	Data transfer/s	GB/s	Module Type
DDR2- 400	100 MHz	200 MHz	10 ns	400MT/s	3.2	PC2- 3200
DDR2- 800	200 MHz	400 MHz	5 ns	800MT/s	6.4	PC2- 6400

Figura 4.15: tabella dei dati di alcune DDR.

4.5.6 DRAM controller



Il compito principale del DRAM controller è quello di prelevare i segnali dal BUS di CPU e di tradurli in segnali di temporizzazione per il modulo che realizza la RAM. Il blocco di refresh non è sempre presente all'interno del DRAM controller, ad esempio le ultime versioni delle memorie hanno questo blocco internamente. Il blocco MPX (MultiPleXer), riceve in ingresso gli indirizzi dall'address BUS e li multiplexa.

Esempio:

In ingresso al MPX vengono mandati 20 bit di ABUS (10 + 10), questo blocco fa sì che i 10 bit di uscita coincidano con gli opportuni segnali di RAS e CAS.

Infine il blocco di TIMING & CONTROL genera i segnali di controllo e temporizzazione, quindi il segnale di RAS, CAS, W/R e il segnale di ready.

4.5.7 Schema di memoria DRAM

Ipotesi:

- Microprocessore Pentium, DBUS a 32 bit, ABUS a 32 bit + BE0-3
- Memoria totale da 1 GB
- Chip da 0.5 Gb, 64MB x 8
- Non si considera rilevazione errori

Ne deriva:

- Quattro rank da 256MB, costituiti da 4 chip da 64M x 8bit
- Un DBUS da 32 bit = 4 byte implica 4 banchi di memoria
- Chip da 64M implicano 26 bit di indirizzo.

L'address BUS sarà quindi suddiviso come segue:

- ABUS 0-1: Per la selezione di uno dei 4 banchi (BE), ossia il banco di partenza da cui leggere il dato.
- ABUS 2-27: Bit necessari per selezionare la parola all'interno del rank.
- ABUS 28-29: Bit necessari per selezionare il rank

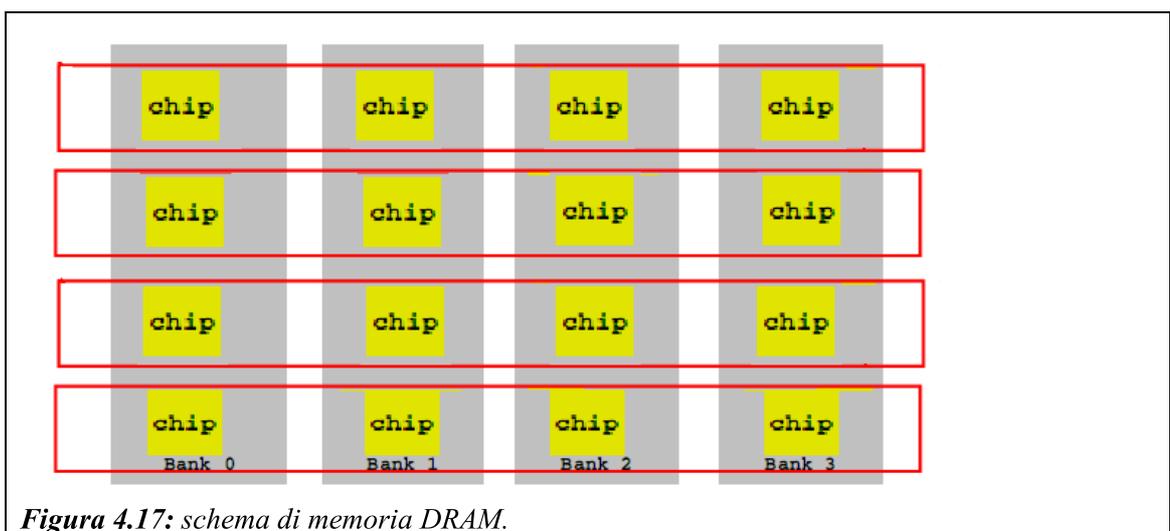
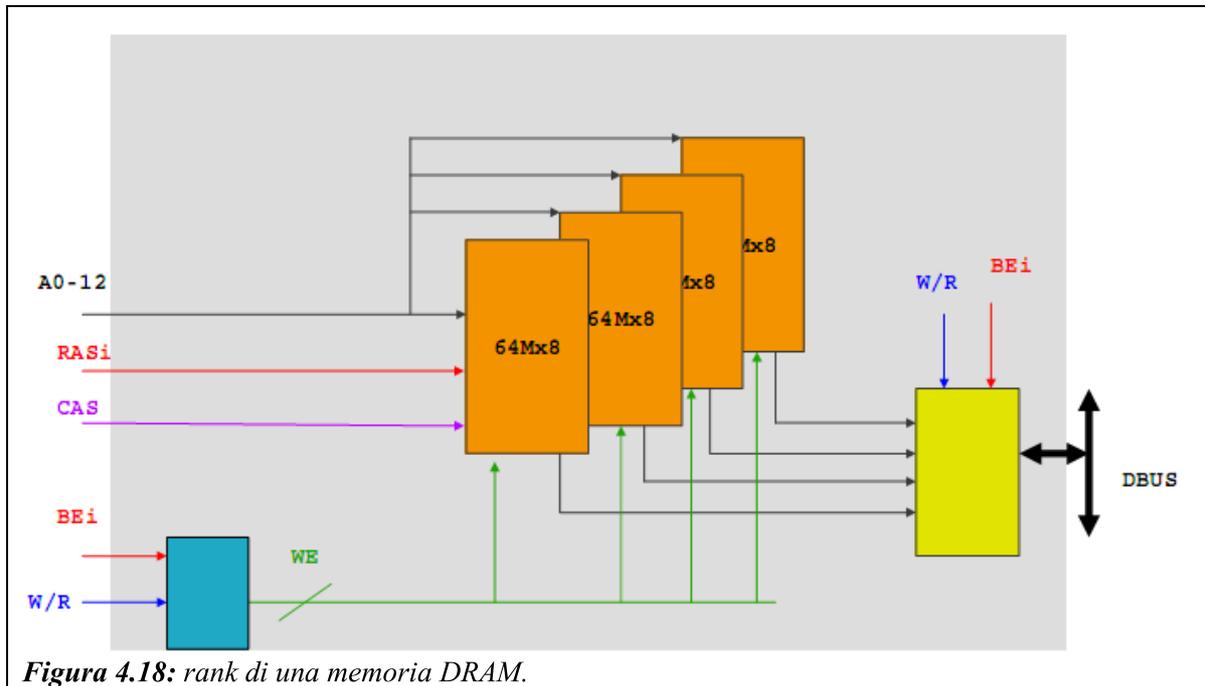


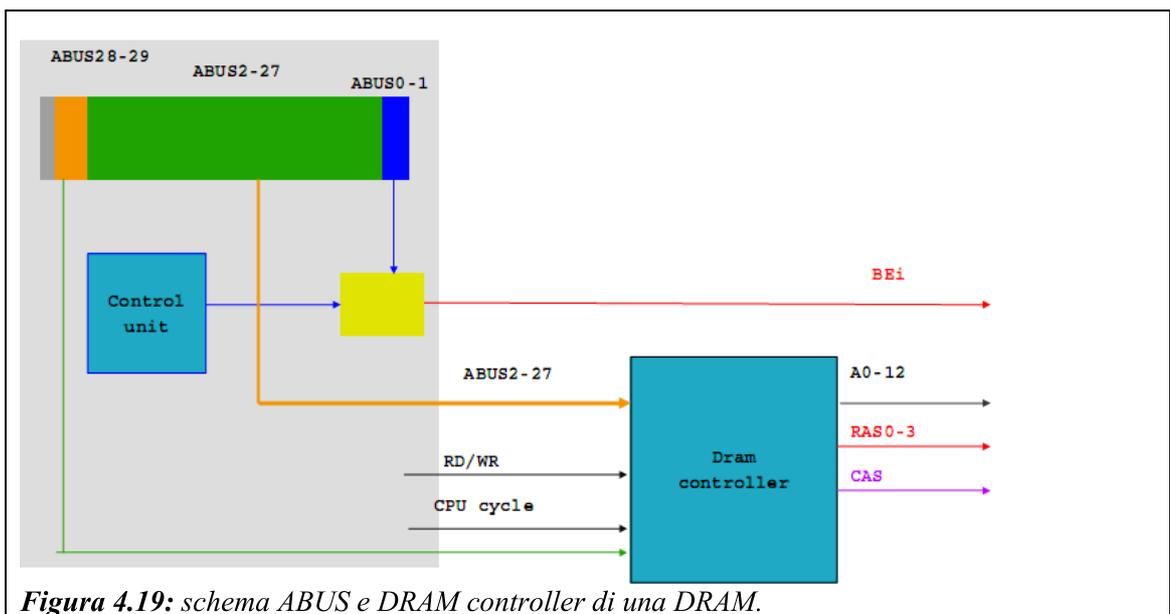
Figura 4.17: schema di memoria DRAM.

Dove ogni rank è organizzato come segue:



Il blocco giallo è un buffer, che ha una funzione simile a quella della coda di prefetch, cioè memorizzare i dati e tenerli pronti per quando vengono richiesti.

Il seguente schema illustra come l' ABUS "comunica" con la memoria:



L'indirizzo che passa nell' address BUS viene "diviso", o meglio ognuno dei singoli blocchi riceve in ingresso i bit necessari per svolgere il suo compito. Il blocco giallo, oltre ai bit dell'ABUS, riceve in ingresso uno o più segnali relativi all'istruzione che si sta eseguendo, i quali condizionano il BE, poiché l'ABUS contiene solo l'indirizzo del banco di partenza, quanti banchi attivare è determinato dall'istruzione in base a quanti byte devono essere trasferiti.

Alla fine unendo tutti i singoli pezzi analizzati la memoria avrà uno schema del tipo:

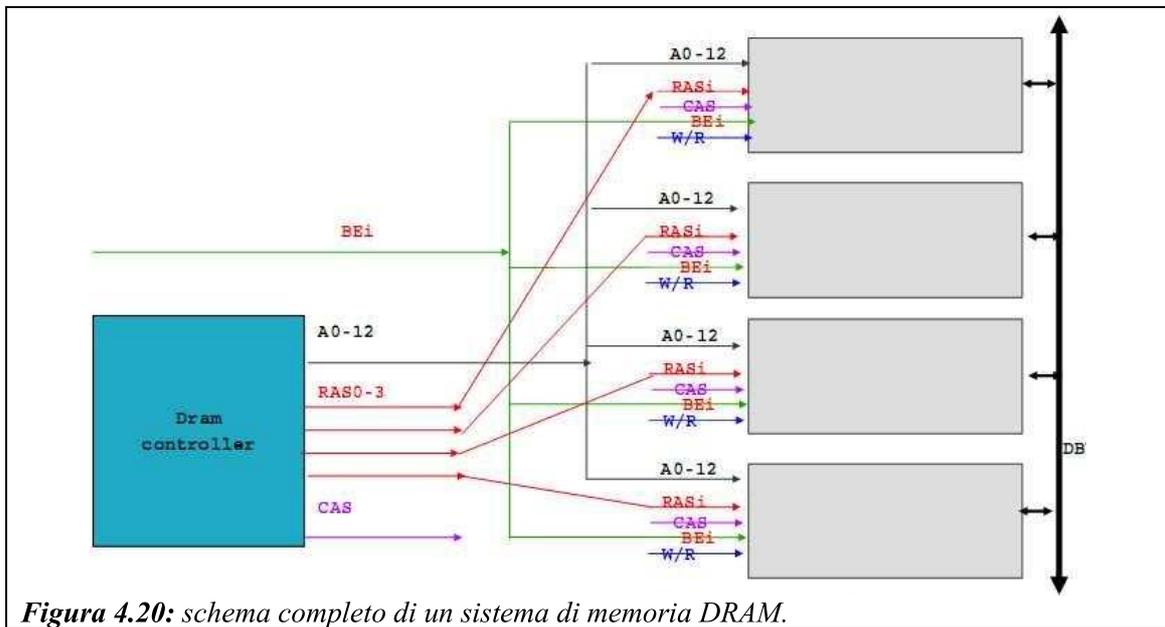


Figura 4.20: schema completo di un sistema di memoria DRAM.

4.6 Controllo d'errore

Quando si parla di memoria, bisogna considerare la possibilità di errori all'interno della stessa.

Gli errori sono di due tipi:

- Transitori: causati da interferenze d'ambiente (ad esempio di tipo elettromagnetico).

Questo tipo di errori sono recuperabili

- Permanenti: causati da difetti hardware (ad esempio una saldatura che si stacca). Questo tipo di errori invece non sono recuperabili

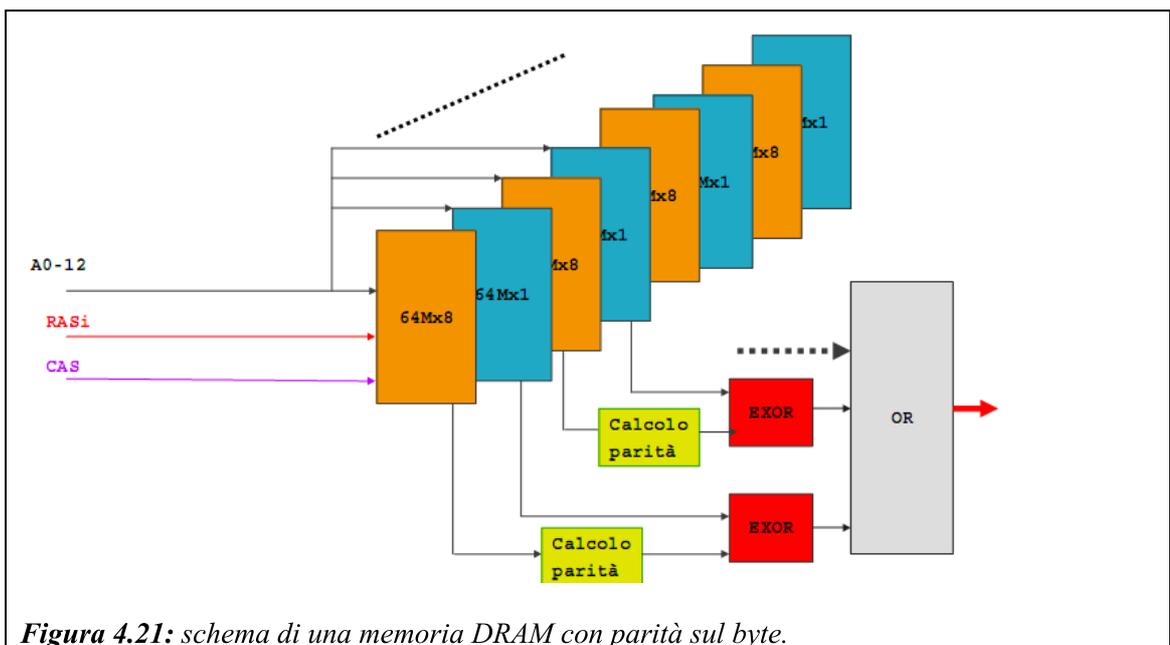
Sono state fatte delle statistiche, dalle quali deriva che:

- L'errore singolo ha alta probabilità
- L'errore doppio ha bassa probabilità
- L'errore triplo ha bassissima probabilità

Come logico pensare, quindi, si tende a rilevare e correggere gli errori che hanno probabilità maggiore. Esistono oggi due schemi per il controllo d'errore:

1. Rilevazione errore di parità sul byte (1bit x ogni byte), oggi i circuiti sono integrati nel processore
2. Rilevazione/correzione errori sul parallelismo di parola (BUS), ECC (8 bit x 8 byte): vengono corretti gli errori singoli e rilevati errori doppi e certi multipli. Comporta una diminuzione delle prestazioni dell'ordine del 2%

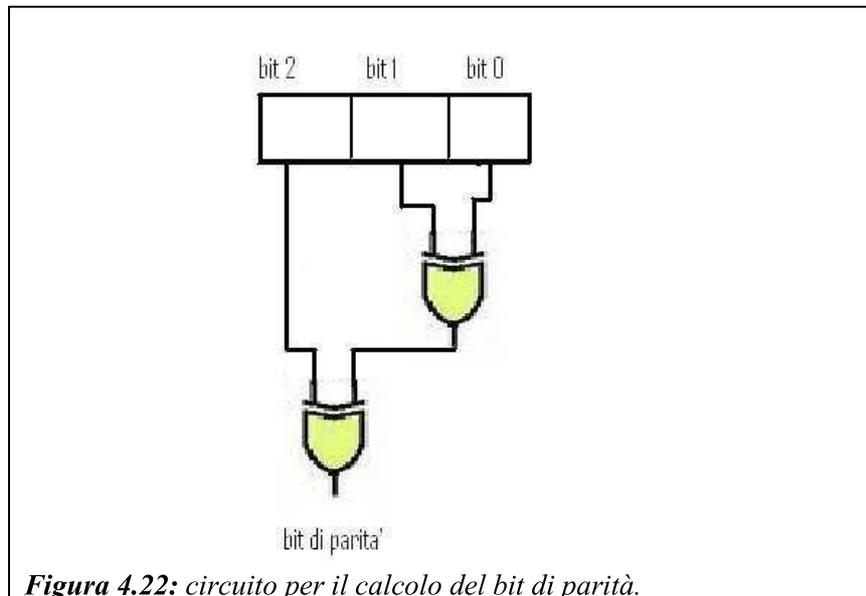
Rank (da 256MB) di memoria DRAM con parità sul byte:



dove i chip blu con dimensione 64M x 1b sono i chip nei quali vengono memorizzati i bit di parità.

Il circuito di calcolo della parità è un circuito composta da una schiera di EXOR.

Esempio calcolo del bit di parità su 3 bit:

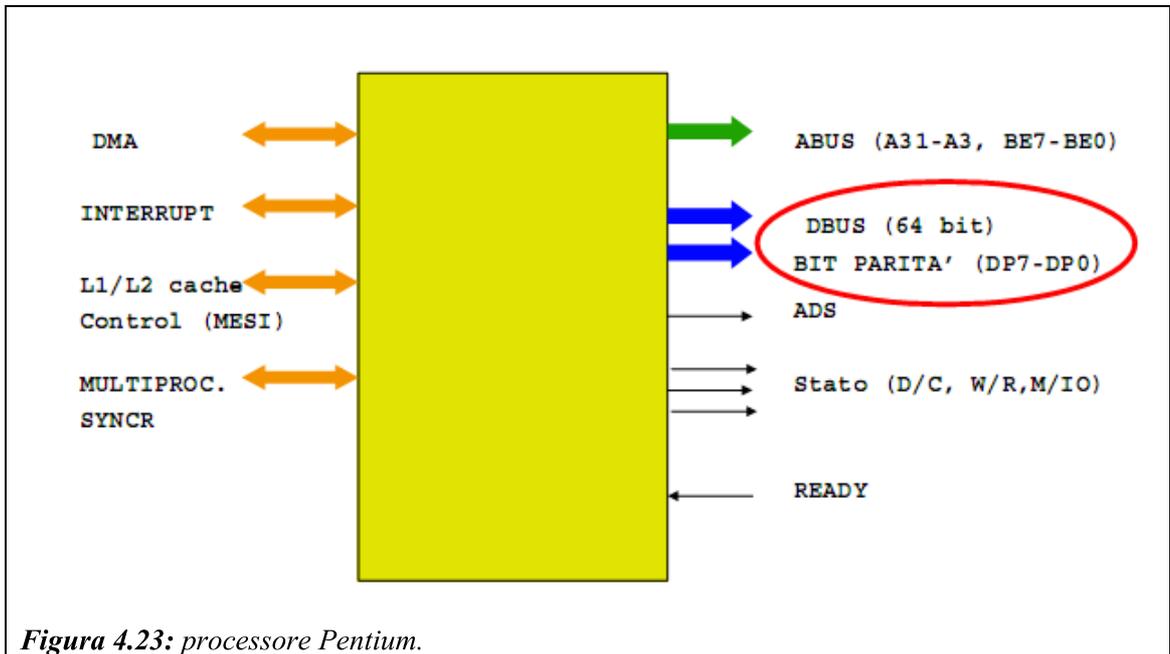


Se si esegue un'operazione di scrittura, i dati vengono mandati sia al chip a cui sono destinati sia al circuito che calcola il bit di parità. Il bit di parità è memorizzato all'interno del chip di parità con la stessa tempistica con cui è memorizzato il dato.

Se si esegue invece un'operazione di lettura, il dato viene prelevato dal chip in cui si trova, ma non viene mandato dritto sul DBUS, viene invece fatto passare nel circuito per il calcolo della parità (lo stesso per operazione di lettura e scrittura). Viene prelevato il relativo bit di parità (calcolato in fase di scrittura), i due bit (quello calcolato in fase di scrittura e quello calcolato in fase di lettura) vengono comparati, se uguali il dato è integro altrimenti il dato non può essere considerato affidabile e viene implementato un meccanismo per la correzione dell'errore. Questo tipo di memorie sono quindi SEC (Single Error Correction) DED (Double Error Detection).

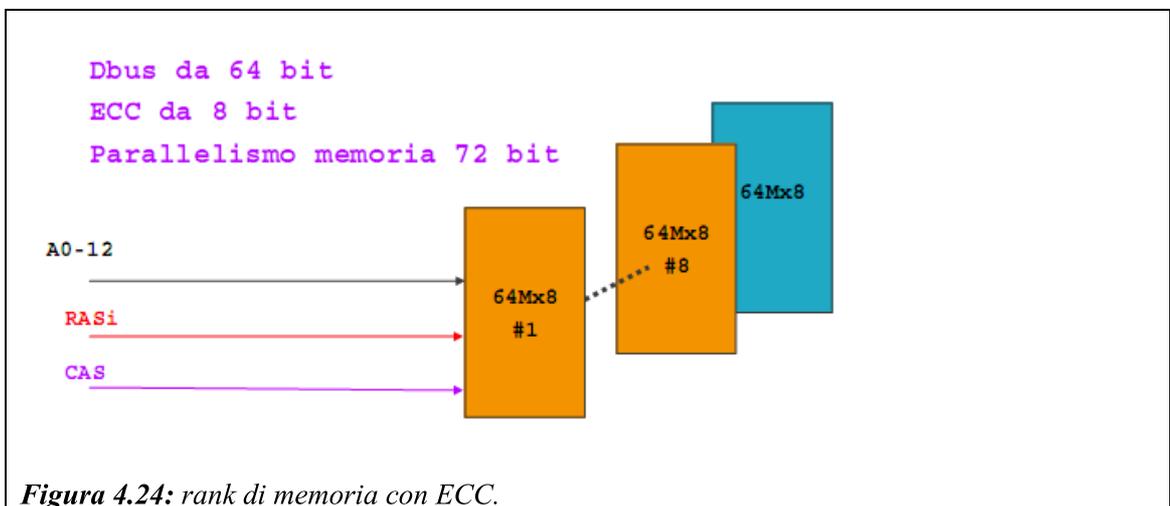
La maggior parte dei processori odierni ha circuiti integrati in grado di generare il bit di parità.

Ad esempio nella famiglia PENTIUM:



si nota la presenza di un DBUS a 64 bit dove i bit che vanno dal bit 0 al bit 7 sono i bit di parità.

Rank (da 512MB) con bit di ECC (Error Correction Code):



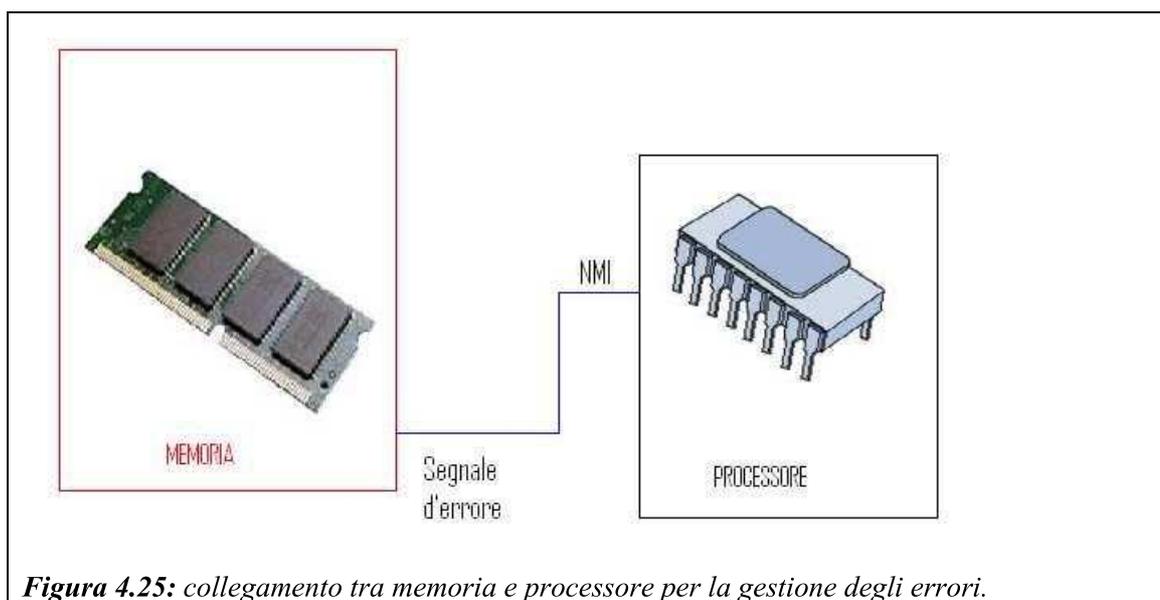
Le RAM con ECC hanno dei sistemi utili a rintracciare eventuali errori contenuti nell'informazione memorizzata e dei meccanismi capaci di correggere l'errore riscontrato. Questo è

possibile registrando informazioni aggiuntive che rendono queste memorie più costose e poco più lente delle rispettive RAM non dotate di ECC (precedentemente descritte). Gli eventuali bit che presentano errori vengono individuati e corretti all'istante senza influire con le applicazioni in esecuzione.

Si può schematizzare il funzionamento nel seguente modo:

per prima cosa occorre controllare se l'errore che è stato rilevato e se è correggibile oppure no. In caso di errore correggibile, questo viene corretto all'interno della memoria stessa senza che il sistema se ne renda conto, se invece l'errore non è correggibile il sistema ne prende atto e vengono attuate delle procedure prestabilite.

Schema semplificato di come vengono attivate queste routine:



Il segnale che esce dalla memoria è un segnale ad altissima priorità; normalmente viene mandato al piedino NMI (Interrupt Non Mascherabile). A questo punto il problema è quale routine attivare e cosa questa deve fare. Per prima cosa il sistema controlla che la routine stessa non sia affetta da errore. Per rendere quasi nulla la probabilità che la routine sia affetta da errore, questa viene

memorizzata in EPROM che essendo una memoria a sola lettura è meno soggetta a errore rispetto ad altri tipi di memoria. A questo punto cosa fare varia da sistema a sistema. Prendiamo il caso dei PC, la procedura prevede la disabilitazione di qualsiasi applicazione lavori con quei dati (compreso il S.O.), a questo punto il PC dà un messaggio di errore e si ferma lasciando all'utente il compito di gestire questa situazione. Questa soluzione è un po' drastica, esistono infatti altre soluzioni chiamate di degradazione morbida.

Esempio:

si hanno vari programmi in esecuzione ognuno dei quali sta lavorando con una porzione di memoria.

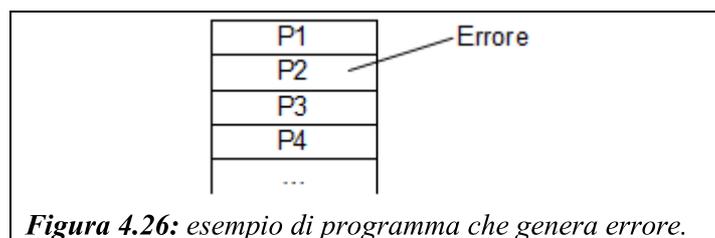


Figura 4.26: esempio di programma che genera errore.

A questo punto P2 subisce un abort, e si deve controllare quale è la pagina di memoria che stava utilizzando, cioè quella in cui è presente l'errore. Questo tipo di gestione dell'errore prevede una riconfigurazione della macchina, cioè una riconfigurazione della memoria. Si toglie la porzione di memoria contenente l'errore e si sostituisce con una porzione di memoria funzionante.

Da sottolineare il fatto che l'implementazione di questo tipo di routine è molto complicata.

4.7 Esercizio relativo al progetto di una memoria

Si vuole realizzare un modulo di memoria di capacità complessiva 2GB e con controllo di parità sul byte. Si dispone delle seguenti informazioni:

- DBUS = 32 bit
- I chip disponibili sono da 128M x 4b

Soluzione:

Un DBUS da 32 bit = 4 byte implica che la memoria deve avere 4 banchi (e quindi 4 segnali di BE), mentre il parallelismo dei chip (4 bit) implica che ci saranno 2 chip per ogni banco. 2 chip x 4 banchi = 8 chip per ogni rank. A questo punto possiamo calcolare la capacità di un rank che è: $8 \times (128\text{M} \times 4 \text{ bit}) = 512\text{MB} \rightarrow 4 \text{ rank}$ per arrivare a 2GB.

A questo punto si illustra come si gestisce il controllo di parità sul byte: si ricorda che il parallelismo dei chip è di 4 bit, si hanno 4 banchi quindi 4 byte ne consegue 1 chip per ogni rank per un totale di 4 chip per il controllo di parità.

La memoria avrà una struttura del tipo illustrato in figura 4.27.

L'ABUS sarà suddiviso nel seguente modo:

- I bit AB0 e AB1, insieme al codice operativo dell'istruzione determinano il banco (o i banchi) da selezionare
- I bit AB2-AB29 (27 bit), indirizzano i 128Mb all'interno del chip
- I bit AB30 e AB31 individuano il rank di appartenenza dei dati.

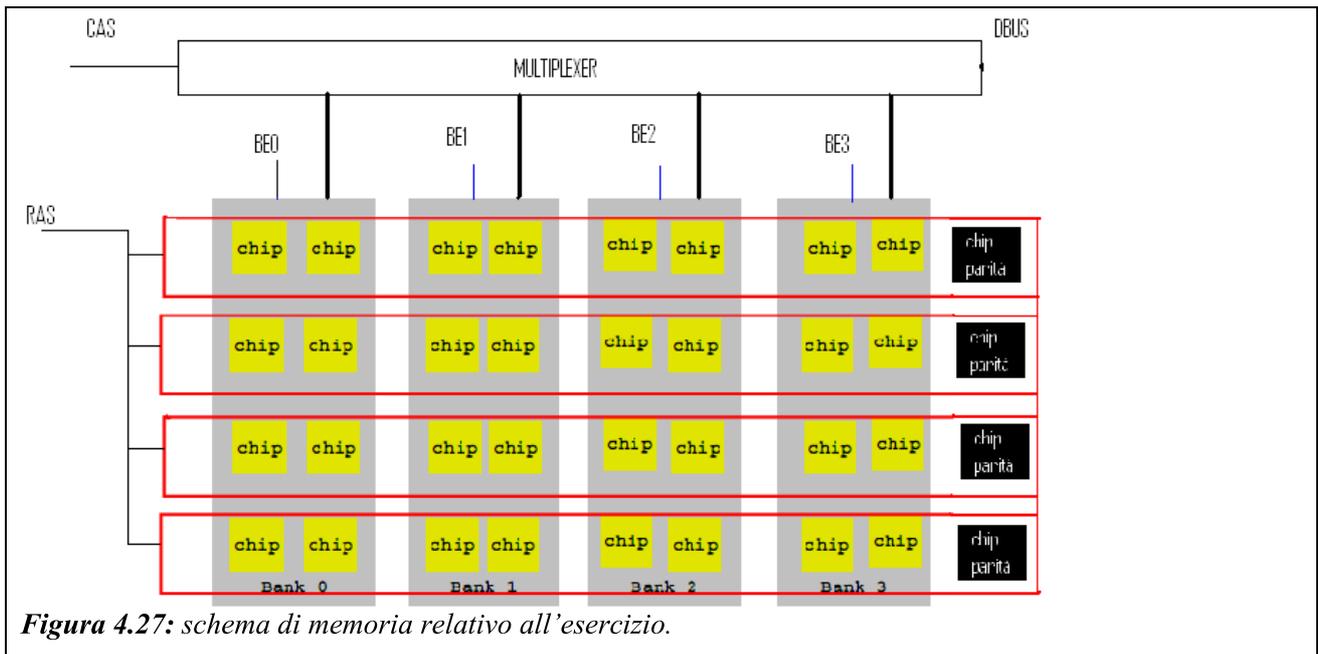


Figura 4.27: schema di memoria relativo all'esercizio.

(Per ragioni grafiche non sono stati disegnati tutti i pin che sono riassunti in un'unica linea, e il DRAM Controller)

I chip di parità vengono correlati con il resto della memoria secondo lo schema descritto al punto 4.6. In maniera schematizzata si può dire che il collegamento tra un chip di parità e il resto della memoria è il seguente (per semplicità si prendono in esame solo 2 chip):

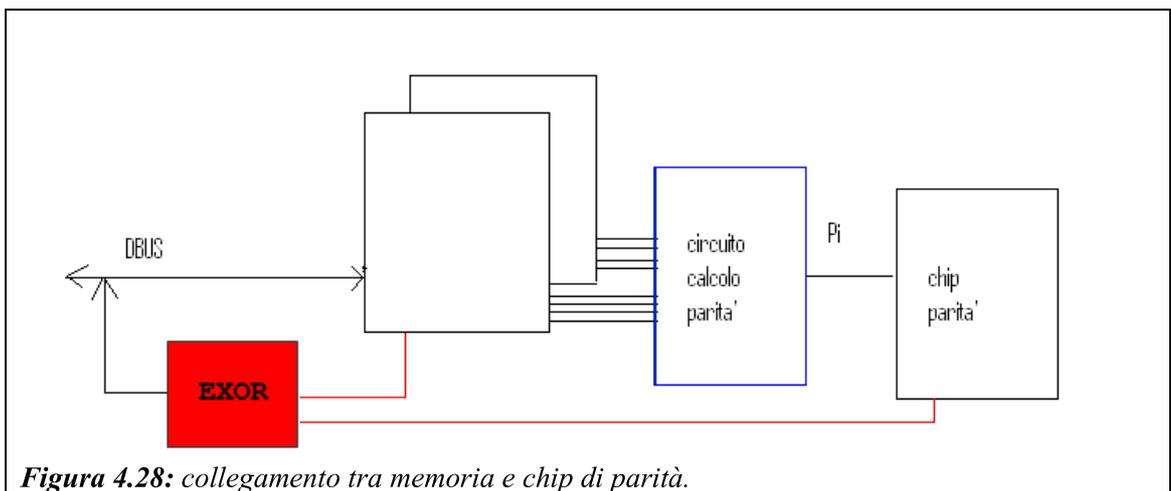


Figura 4.28: collegamento tra memoria e chip di parità.

Due chip hanno un'uscita complessiva di 8 bit (1 byte), il comportamento è uguale a quello descritto nel punto 4.6 (controllo d'errore).

Riferimenti

IA-32 Intel® Architecture Software Developer's Manual Volume 3: System Programming Guide

Slide. Architettura x86 – 32; Prof. Marco Mezzalama – a.a. 10/11

Slide. Bus di sistema e di cpu nell'architettura 80x86; Prof. Marco Mezzalama – a.a. 10/11

Slide. Sottosistema di memoria nell'architettura 80x86; Prof. Marco Mezzalama – a.a. 10/11

Siti internet consultati

http://it.wikipedia.org/wiki/Intel_8086

http://it.wikipedia.org/wiki/Error-correcting_code

<http://www.ingmonti.it/libri/Parte2/index.html>

http://it.wikipedia.org/wiki/Pipeline_dati

http://it.wikibooks.org/wiki/Architetture_dei_processori/Architettura_Pentium_4

<http://it.wikipedia.org/wiki/Boot>

Videolezioni

Lezione 1. Prestazioni degli elaboratori; Prof. Marco Mezzalama – a.a. 10/11

Lezione 2. Sistemi emicroprocessori; Prof. Marco Mezzalama – a.a. 10/11

Lezione 3. Architettura x86 reale; Prof. Marco Mezzalama – a.a. 10/11

Lezione 4. Pipeline; Prof. Marco Mezzalama – a.a. 10/11

Lezione 13. Architettura della memoria (1/2); Prof. Marco Mezzalama – a.a. 10/11

Lezione 14. Architettura della memoria (2/2); Prof. Marco Mezzalama – a.a. 10/11

Lezione 15. Controllo di errore nelle memorie; Prof. Marco Mezzalama – a.a. 10/11