L'ISA x86

Prof. Alberto Borghese
Dipartimento di Informatica
alberto.borghese@unimi.it

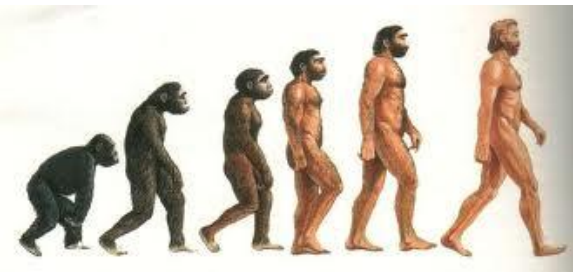
Università degli Studi di Milano

Patterson, sezioni 1.13, 2.19, 4.12

Sommario

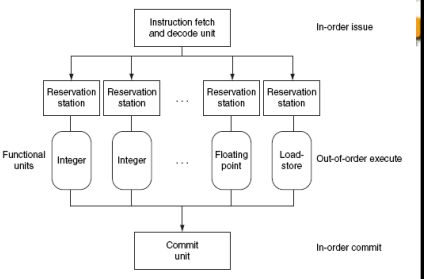
Evoluzione degli Intel

La compatibilità backwards ha costretto la ricerca di soluzioni innovative.



Dall'8080 al 386, al Pentium, ai multi-core...

Le pipeline Intel



Ultimo processore Intel senza pipeline: 386, 1985.

UC **cablata** (cf. MIPS) per le istruzioni semplici e **microprogrammata** per le istruzioni più complesse (esegue sequenze di operazioni elementari).

Pentium 4: Superpipeline superscalare. Fino a 3 istruzioni per ciclo di clock.

Core i7. Sperimentalmente in media 6 istruzioni per ciclo di clock.

Trasformo le istruzioni dell'ISA Intel in micro-operazioni di lunghezza uguale (RISC) – gli issue sono costituiti da 3 micro-operazioni**.**

A partire dal codice operative dell'istruzione, vengono generati i segnali di controllo: 120 per le ALU intere, 275 per le unità ALUfp e 400 per le istruzioni SS2.

Micro-architettura del Core i7 920

Internal micro-operations RISC

Multiple-issue (cammini paralleli di esecuzione)

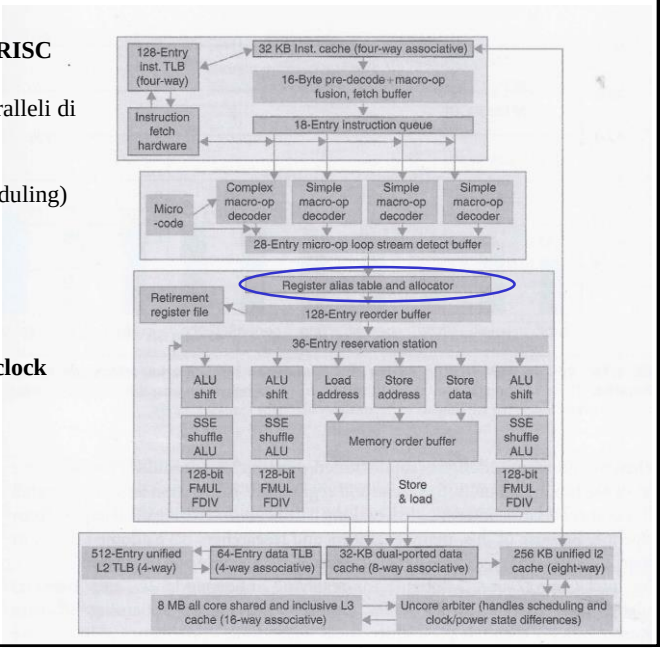
Superscalari (dynamic scheduling)

Speculazione + roll-back

Register renaming

6 micro-operazioni / ciclo_clock
Esecuzione fuori ordine.

14 stadi di pipeline
8 fasi principali



Fase di fetch - I

1) **Fetch - I**

a. Lettura: dalla cache di secondo livello alla instruction cache (32 Kbyte).

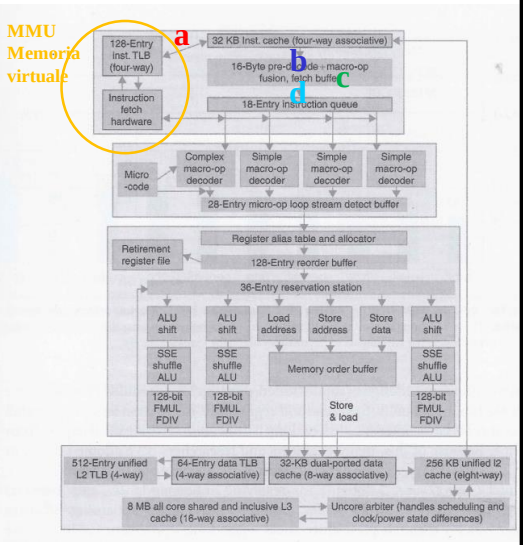
b. Trasferimento di gruppi di 16 Byte nel buffer di pre-decodifica

- Utilizzo di un BPB multi-livello.
- Misprediction: 15 cicli di clock.

2) **Fetch - II**

c. Identificazione delle istruzioni Intel nel buffer di pre-decodifica (1-15 Byte)

d. Inserimento delle istruzioni Intel nella coda di decodifica (18 elementi)



Fase di decode - II

3) Decodifica

a. Le istruzioni x86 vengono tradotte in macro-operazioni

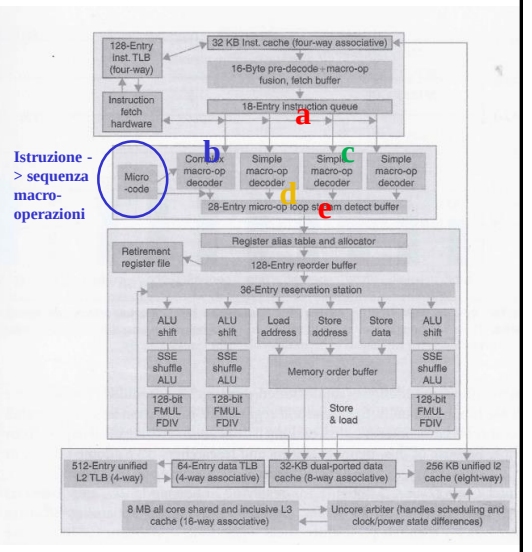
b. Un'istruzione x86 complessa viene trasformata in una sequenza di micro-operazioni

c. Un'istruzione x86 semplice viene trasformata direttamente in una micro-operazione per l'esecuzione.

d. Riempimento della coda di dispatch delle micro-operazioni (per lo scheduler).

4) Trattamento dei cicli

e. Loop unrolling (28 istruzioni max, 256 Byte),



Fase di esecuzione - III

5) Scheduling

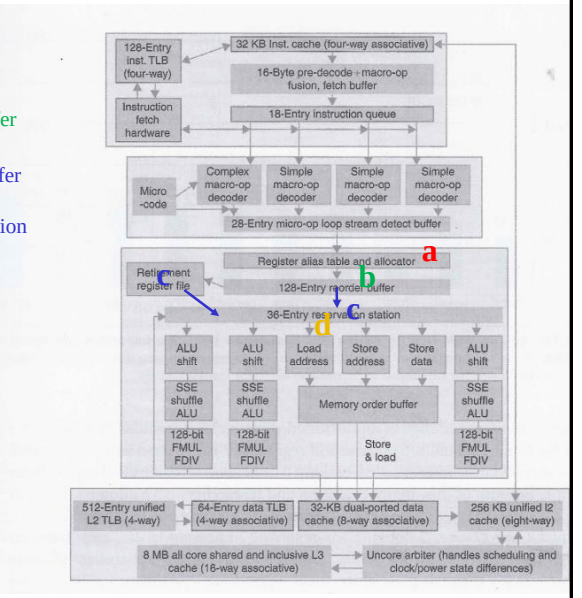
a. Register renaming (mappatura)

b. Allocazione di spazio nel reorder buffer per il risultato

c. Lettura degli operandi dal reorder buffer (propagazione) o dal Register File e inserimento degli issue nelle Reservation Station.

6) Launch execution

d. Multiple-issue costituito da 6 issues avviati ad esecuzione



Fase di write-back (commit) - IV

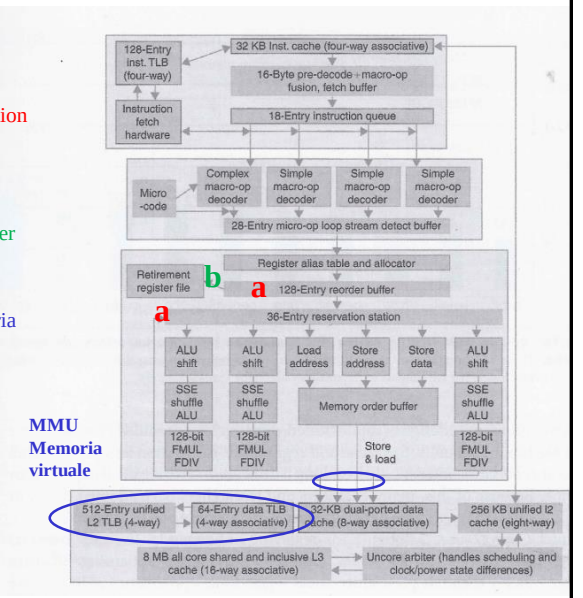
7) Termine esecuzione

a. Il risultato viene restituito nel reorder buffer e, se serve, inviato alla reservation station (feed-forward)

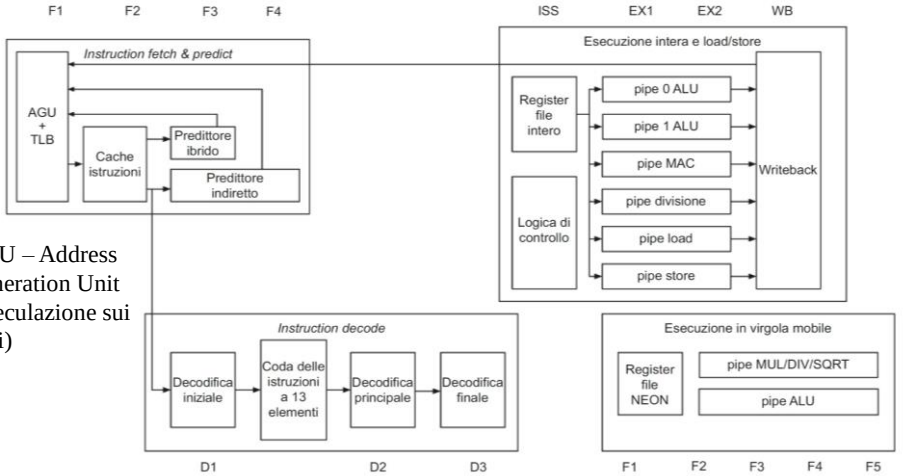
8) Write-back

b. Risultato copiato nel retirement register file quando non è più speculativo.

Store e load si interfacciano con la memoria dati.



Micro architettura Cortex A53 ARM

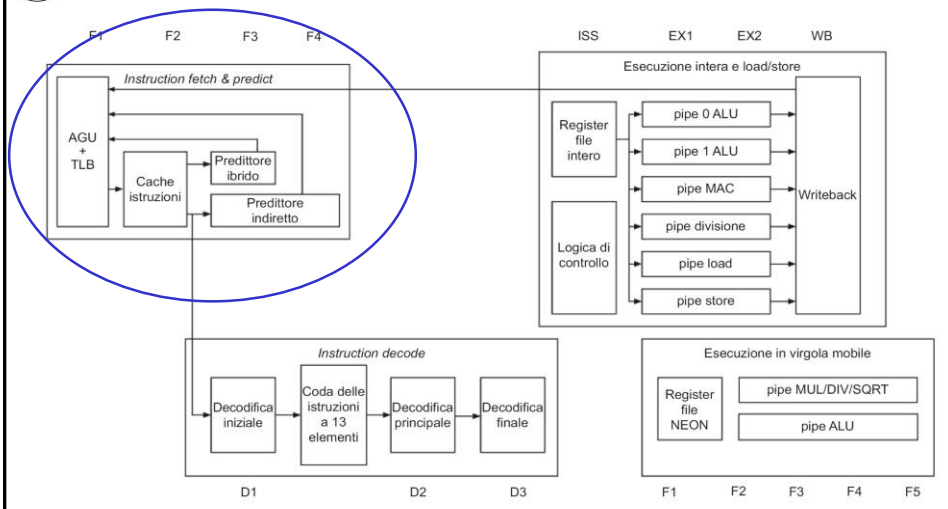


AGU – Address Generation Unit (speculazione sui salti)

Core delle architetture x tablet e smart phone

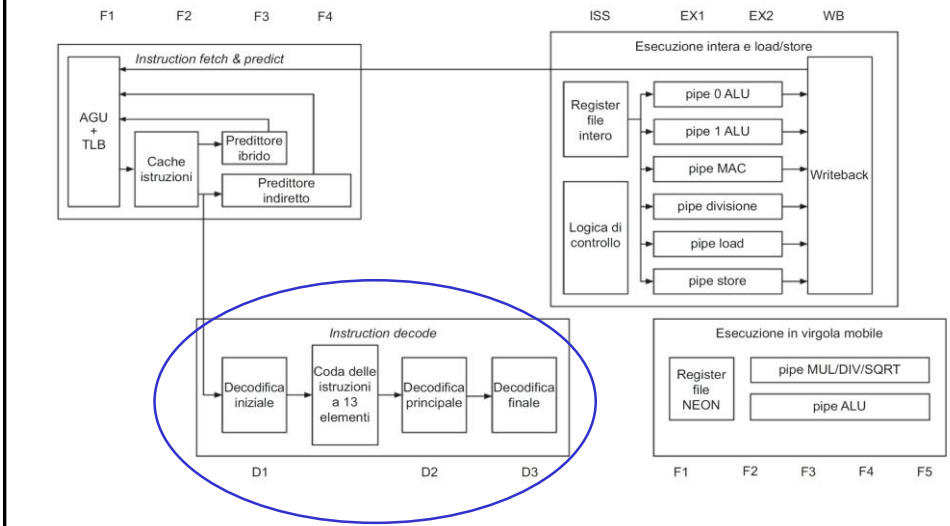
- Schedulazione statica (2 cammini di esecuzione VLIW di 2 istruzioni)
- Istruzioni intere passano attraverso 8 stadi: F1, F2, D1, D2, D3 / ISS, EX1, EX2 e WB
- Serializzazione della coppia di istruzioni in caso di hazard.

Fase fetch - Cortex A53 ARM



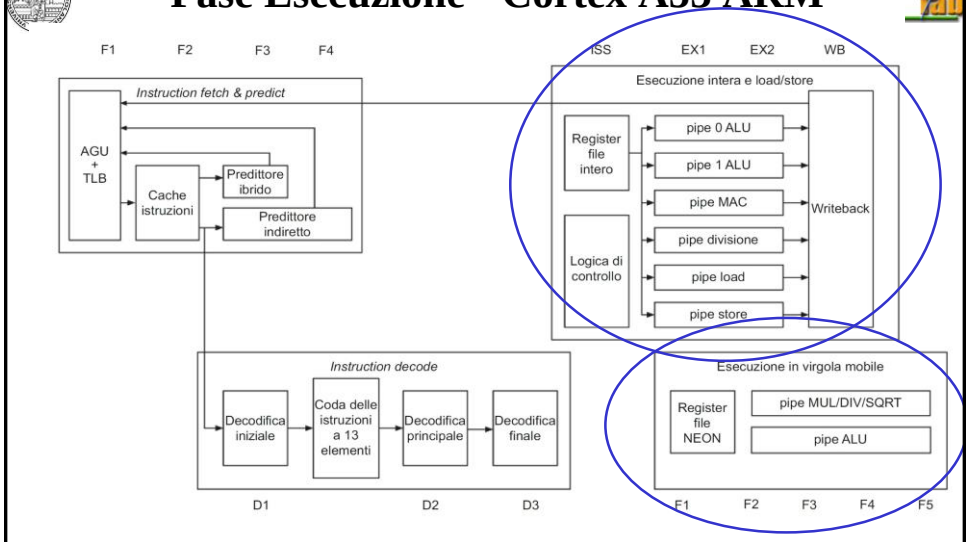
Cache destinazione dei salti (2 istruzioni, 1 elemento, coppia successiva all'istruzione di branch)
Predittore ibrido (cache di 3072 elementi, nel caso in cui la coppia di istruzioni non sia quella richiesta, 2 cicli di clock).
Predittore indiretto (256 elementi, ritardo di 3 cicli di clock).

Fase decodifica - Cortex A53 ARM



Decodifica di istruzioni semplici (D1 e D2)
Decodifica di istruzioni complesse (D3). Questa fase si sovrappone alla prima fase di esecuzione (ISS)

Fase Esecuzione - Cortex A53 ARM



Pipe 0 – salti condizionati + istruzioni su interi
Pipe 1 – istruzioni su interi
Pipe separate per load/store (collegamento tra WB e register file)
L'unità funzionale NEON esegue le istruzioni SIMD ARM che iniziano con lo stesso nome.

ARM Cortex-A53 e Intel Core i7 920

Processore	ARM 53	Intel Core i7
Mercato	Dispositivi mobili	Server, Cloud
Obbiettivi di consumo	2 W	130 W
Frequenza di clock	1 GHz	2,66 GHz
Core / chip	1	4
Virgola mobile?	Sì	Sì
Multiple-issue	Statico	Dinamico
ICP di picco	2	4
Stadi pipeline	14	14
Scheduling pipeline	Statico In-order	Dinamico Out-of-order con Speculazione
Predizione salti	A 2 livelli	A 2 livelli
Cache 1° livello	32 KB I; 32 KB, D	32 KB I; 32 KB, D
Cache 2° livello	128 – 1024 KB	256 KB
Cache 3° livello	-	2-8 MByte

Sommario

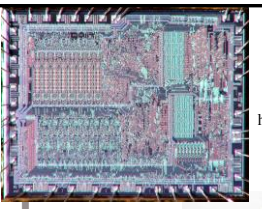
Le architetture Intel

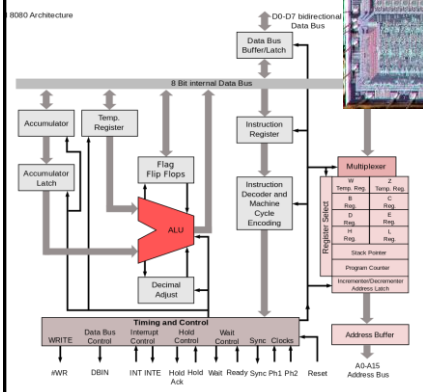
Evoluzione degli Intel

L'ISA x86

La codifica delle istruzioni

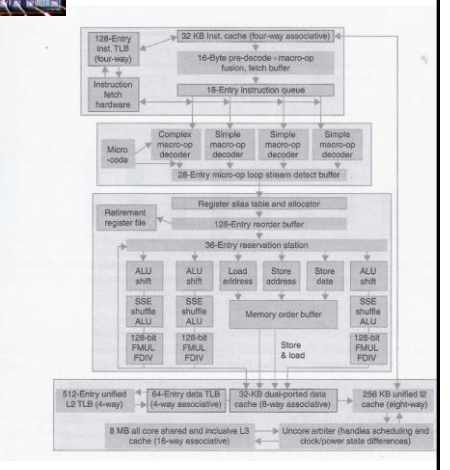
Intel Core i7





https://commons.wikimedia.org/wiki/File:Intel_8080_arch.svg

https://www.cpu-world.com/CPUs/8080/



INTegrated ELectronics (1968), mainly memories.
Evoluzione dell'Intel 8086 (1978)
Simile all'evoluzione delle speci
Compatibilità a ritroso

"This checkered ancestry has led to an architecture that is difficult to explain and impossible to love" [Patterson Hennessy

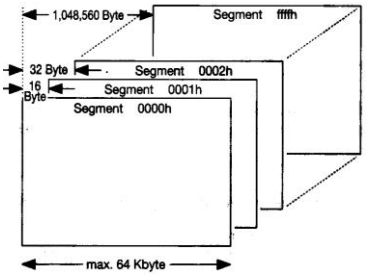
Le prime architetture Intel

- **1978 – 8086**
  - Estensione del micro-processore 8080 utilizzato per applicazioni industriali (micro-controllore). Stessa ISA. **Architettura a 16 bit** con registri a 16 bit. Parte dei registri è dedicata a compiti specifici, **non è un'architettura con registri general-purpose**.
- **1980 – 8087**
  - Coprocessore matematico. Dedicato a velocizzare le operazioni in virgola mobile. Modifica nel modo di gestire gli operandi. Questi potevano essere **presi dallo stack (virgola mobile sempre) oppure dai (pochi) registri.**
  - Estensione degli operandi a 10 byte (80bit), *Extended Double Precision*.
    - E' il compilatore a potere dichiarare variabili su 10byte (Extended Double).
  - Ciclo di esecuzione di un'operazione aritmetica (dati in stack):
    - Push <operando_1> in stack (esteso a 10byte); Push <operando_2> in stack (esteso a 10byte).
    - Operazione.
    - Pop <risultato>da stack

- Limite nello spazio di indirizzamento: 1 Mbyte (2^{20}).
  - Indirizzamento operato come: `Base shl 4 + Offset`
- Gestione di memoria ed I/O tramite 3 segnali di controllo: **RD, WR, IO/MEM**

Indirizzamento in modalità reale

Interleaving dei segmenti:
Spazio di indirizzamento di 1Mbyte suddiviso in segmenti di 64kbyte



- Modalità reale
  - Indirizzo = 16 * Segmento + Offset
  - In tal modo posso indirizzare 64KB x 16 = 1 MByte di MM
  - Modalità Virtual 8086: Indirizzo e offset sono separati.
    - CS = 0x80B8, IP = 0x019D
      → indirizzo istruz. successiva:

0x 80B8 0 +
019 D =

0x 80D1 D

Gestione della memoria a segmenti

Pagine: ***segmenti di dimensioni fisse.***

Molto architetture (a 64 bit) dividono lo spazio di indirizzamento in blocchi di pagine contigue di grandi dimensioni che vengono chiamti "segmenti", in modo improprio.

I segmenti hanno:

- Gli indirizzi fisici non vengono individuati in modo univoco (diverse combinazioni di indirizzo di segmento + offset) -> Estensione dello spazio di indirizzamento (complessità eccessiva, oggi abbandonata).
- ***Dimensione variabile (nelle architetture attuali)***

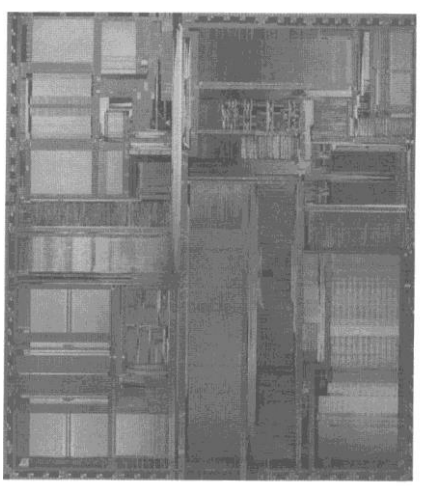
Complessità per la protezione della memoria (numero di segmento + offset)

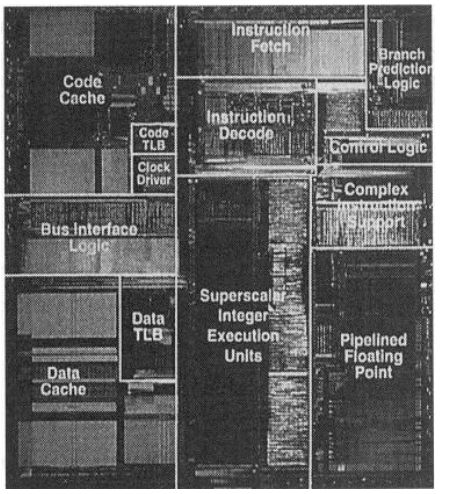
- Segmenti di dimensioni diverse -> offset può portare al di fuori di un segmento -> boundary check.

Le architetture Intel avanzate

- **1982 – 80286**: **L'architettura diventa a 24 bit**. Viene utilizzata una modalità di utilizzo protetta che consente di mappare le pagine di memoria in indirizzi privati. Aggiunta di istruzioni specifiche.
- **1985 – 80386**: **Estensione a 32 bit.** Archtettura, spazio di indirizzamento e registri a 32 bit. Nuove istruzioni, molto vicino ad un calcolatore con general-purpose registers. **Pre-fetching.** Paginazione della Memoria Principale.
- **1989-1992 – 80486**: Istruzioni per il trasferimento di dati condizionato. Cache. **Pipe-line singola.** Microprogrammazione per l'Unità di controllo (FSM).
- **1992-1995 – Pentium, PentiumPro**: Pipe-line multiple (super-scalare). Cache primaria e secondaria (separata, con bus dedicato). Introduzione dell'esecuzione condizionata. ISA quasi identico. Ampia ristrutturazione interna (=> **micro-architettura RISC**: micro-operazioni).
- **1997 – Pentium II**: Memorie cache a doppio accesso per ciclo di clock. Cache dei registri di segmento. PentiumPro + MMX (Multi-media extension, SIMD su dati piccoli: byte).
- **1999 – Pentium III**: "Internet Streaming single instruction multiple data extensions (SSE, **calcolo vettoriale**). Estensione dell'architettura MMX a istruzioni su **dati floating-point**. **L2 cache e CPU nello stesso integrato**. 8 registri da 128 bit per supporto ad esecuzione parallela (multiple-issue).

Il pentium





Esecuzione "speculativa": scheduling dinamico + predizione dei salti (e.g. Intel 80x86 dal Pentium). Esecuzione super-scalare delle operazioni su interi.

Le architetture più recenti

- **2001 – Pentium 4**: Estensione del parallelismo e della Superscalarità. *Hyper-Threading Technology*. SSE2. I registri a 128 bit supportano anche virgola mobile in doppia precisione. **Register renaming e gestione separate delle code di esecuzione.** Massimizzazione delle prestazioni.
- **2003 – AMD**: Architettura a 64 bit. Aumento dello spazio di indirizzamento a 64 bit e registri a 64 bit. **16 registri GP + 16 registri a 128 bit.**
- **2004 – EM64T**: (Extended Memory 64 Techniology). SSE3 che supporta numeri complessi e strutture. **Operazione atomica su blocchi di memoria di 128 bit.**
- **2006 – Virtual machines support:** SSE4 con operazioni quali somma di differenze in modulo, conteggio elementi, ... **Supporto alle macchine virtuali.**
- **2006** – Intel Itanium 2 9000 serie **– Dual Core CPU**.
- **2007 – AMD.** Istruzioni su 3 operandi.
- **2007 – Intel.** Advanced Vector Extension. Registri da 128 a 256 bit.
- **2011 – Intel Core i7 Sandy Bridge. AVE.** Registri SSE fino 256 bit, con modalità di esecuzione vettoriale, **integrazione GPU**.
- **2013 – Canali multipli** per il trasferimento dalla memoria principale.
- **2017 – Intel Core i9**. Supporto alla crittografia.

Trend:

«Fuse add and multiply»

Esecuzione vettoriale

Protezione dei dati

Sommario

Le architetture Intel

Evoluzione degli Intel

L'ISA x86

La codifica delle istruzioni

Architettura x86

- CISC
  - Lunghezza istruzioni: 1 – 15 bytes
  - Operazioni direttamente in memoria
- Architettura condizionata dalla storia → necessità di compatibilità verso il basso
  - Real mode/Protected mode/Virtual 8086 Mode
- Nasce come architettura "not" general-purpose register
  - ogni registro è progettato per un uso specifico
  - dal 80386 in poi (IA-32) si definiscono 8 GP registers, ma non veri GP registers come in MIPS

Modi di funzionamento IA-32

- Modalità reale (Real mode)
  - Modalità compatibile DOS
  - Max memoria indirizzabile: 1 MByte → 2^{20}
  - modo attivo all'accensione (power-on)
- Modalità protetta (Protected mode)
  - modalità "nativa" di IA-32
  - Memoria indirizzabile: 4 GByte → 2^{32}
  - Memoria protetta: evita corruzione memoria da parte di altri programmi
  - Memoria virtuale: permette ad un programma di disporre di più memoria di quella fisica disponibile
- Modalità "8086 virtuale" (Virtual-8086 mode)
  - "Real mode" simulato all'interno del "Protected Mode"
  - Esecuzione di programmi DOS in multitasking con altri programmi che girano in Protected Mode.

Indirizzamento in modalità protetta

- Modalità protetta
  - Spazio di indirizzamento: 32 bit
  - I segmenti sono di fatto pagine di 64Kbyte e non più ogni 16 byte.
    - SALTO: indirizzo = composizione registri CS e IP
    - Esempio:
      - CS = 0x80B8, IP = 0x019D
        → indirizzo istruz. Successiva mediante pipe dei due registri:
        0x 80B8 || 019D

Architetture IA-64: caratteristiche

- 2 tipi di archiettture a 64 bit: x64 (AMD) and IA64 (Intel). x64 è la più utilizzata. E' stata data in licenza a Intel che la ha sviluppata nell'architettura Itanium assieme ad HP.
- Modalità a 64 bit (**long mode**): è compatibile con le Architetture a 32 bit (**legacy mode**).
- Paginazione gerarchica su 4 livelli (***PAE – Physical Address Extension***). Segmentazione ignorata.
- Introduzione di un "**not executable**" bit nello stato di ogni pagina (protezione addizionale).
- Registri a 64 bit -> dati su 64 bit.
- **16 Registri general purpose:** rax, rbx …. + r0 … r 15.
- Istruzioni registro-registro. E' un'architettura load-store a tutti gli effetti. La parte di load-store e' gestita in fase di pre-fetch.
- Le istruzioni sono organizzate in gruppi di istruzioni, **bundle,** con formato FISSO e designazione **esplicita** delle dipendenze (**issue slot**). Sono istruzioni che non hanno criticità (vere) tra loro e che sono separate da un gruppo successive da un separatore (stop).
- Architetture Itanium hanno una dotazione interna ricca di registri (**register renaming**).
- Istruzioni speciali e funzionalità per l'eliminazione dei salti (**loop unrolling**).
- Estensioni vettoriali: registri xmm, ymm, zmm (da 128 a 512 bit).

Bundle IA-64

Le istruzioni sono codificate su 128 bit: **bundle** (VLIW). 2 bundle per ciclo di clock avviati a esecuzione (6 istruzioni / ciclo_clock su 6 cammini).

Ogni bundle ha un template (formato) di 5 bit + 3 istruzioni (ciascuna di 41 bit).

Il formato (template) specifica quale tra 5 differenti unità di esecuzione è richiesta da ciascuna delle 3 istruzioni (M-unit (memory instructions), I-unit (integer ALU, non-ALU integer, or long immediate extended instructions), F-unit (floating-point instructions), B-unit (branch or long branch extended instructions)).

Separatore di stop tra bundle: separa gruppi di bundle senza dipendenze.

Esecuzione predicativa:

if (p) espressione_1

if (not_p) espressione_2

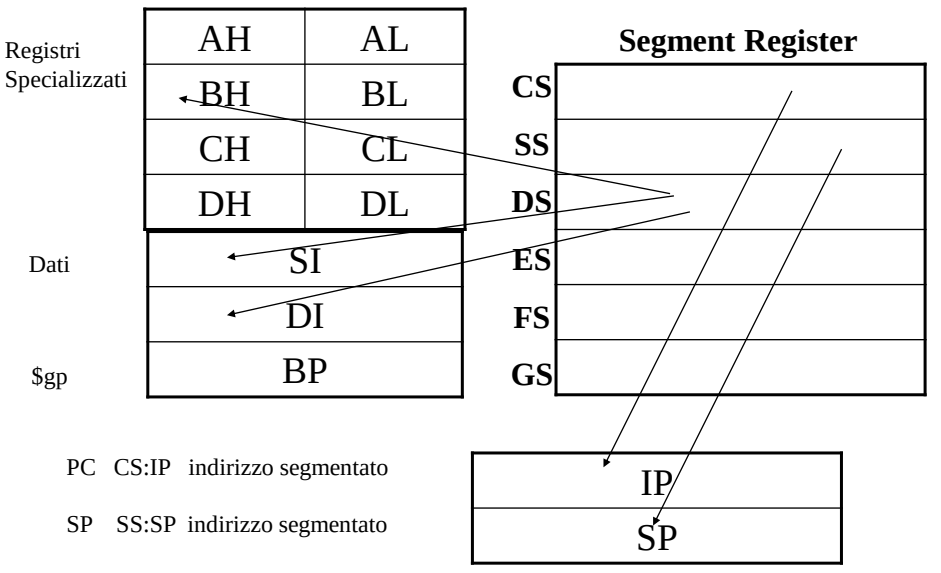
Se espressione_i non deve essere eseguita, il risultato viene disabilitato (l'operazione viene sostituita con una nop) e i risultati nl commit buffer eliminati.

64 bit architectures

Operating mode	Operating sub-mode	Operating system required	Type of code being run	Default address size	Default operand size	Supported typical operand sizes	Register file	Typical GPR width
Long mode	64-bit mode	64-bit operating system or bootloader	64-bit code	64 bits	32 bits	8, 16, 32, or 64 bits	16 registers per file	64 bits
	Compatibility mode	64-bit operating system or bootloader	32-bit protected mode code	32 bits	32 bits	8, 16, or 32 bits	8 registers per file	32 bits
		64-bit operating system or bootloader	16-bit protected mode code	16 bits	16 bits	8, 16, or 32 bits	8 registers per file	32 bits
Legacy mode	Protected mode	32-bit operating system or bootloader, or 64-bit bootloader	32-bit protected mode code	32 bits	32 bits	8, 16, or 32 bits	8 registers per file	32 bits
		16-bit or 32-bit operating system or bootloader, or 64-bit bootloader	16-bit protected mode code	16 bits	16 bits	8, 16, or 32 bits	8 registers per file	16 or 32 bits
	Virtual 8086 mode	16-bit or 32-bit operating system	16-bit real mode code	16 bits	16 bits	8, 16, or 32 bits	8 registers per file	16 or 32 bits
	Real mode	16-bit or 32-bit operating system or bootloader, or 64-bit bootloader	16-bit real mode code	16 bits	16 bits	8, 16, or 32 bits	8 registers per file	16 or 32 bits

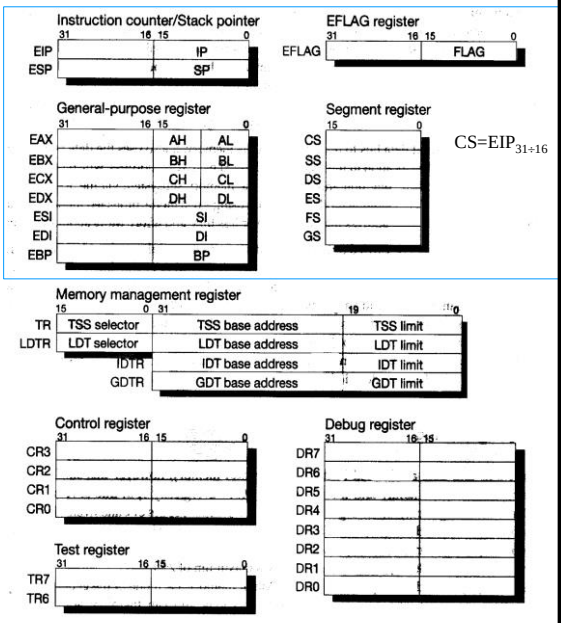
From wikipedia

I registri dell'architettura 80286



I registri dell'architettura IA-32

- Estensione a 32 bit del set dei registri dell'80286.
- E.g. AH -> AX -> EAX
- ISA a partire dal 80386 (IA-32)
- 8 registri "general-purpose" a 32 bit (general puropose + ESP).
  - ma si deve potere accedere ad 1 o 2 byte al loro interno (compatibilità con 8086)
  - non sono poi così "general purpose"…
- I registri di segmento sono rimasti a 16 bit
  - usati come registri base
  - Possono essere utilizzati in alternativa ai registri estesi: (e.g. CS:IP .> EIP, SS:SP -> ESP, DS <-> EBX).



Registri di Segmento (di memoria) a 16 bit

Nome	Descrizione	Utilizzo
CS || IP (EIP)	Code Segment	Puntatore al segmento di codice. Contiene l'indirizzo base dei (dati) ed istruzioni ad accesso immediato. Le istruzioni del segmento sono indirizzate tramite il registro **EIP** (Extended Instruction Pointer). Per modificare il registro **CS** occorre una *chiamata a procedura far* o una *far jump* oppure un interrupt (*int*). In modo protetto viene verificato se il nuovo segmento può essere utilizzato dal task.
DS (EBX)	Data Segment	Puntatore al segmento dati. **Contiene l'indirizzo base del segmento dati del programma ($gp)** Molte istruzioni quali la **mov** utilizzano questo segmento. E' il segmento in cui sono contenuti i dati di un task. In modo protetto viene verificato se il nuovo segmento può essere utilizzato dal task.
SS || SP (ESP)	Stack Segment	Puntatore al segmento di stack. Quasi del tutto simile allo stack del MIPS. Cresce verso il basso. Contiene i dati locali delle procedure e gli argomenti di chiamata. Contiene anche **gli operandi per le operazioni aritmetiche** di tipo accumulatore. In modo protetto viene verificato se il nuovo segmento può essere utilizzato dal task.
ES, FS, GS	Extra Segments	Puntatori addizionali al segmento dati. Principalmente utilizzati per operazioni su stringhe. Possono essere utilizzati in sostituzione di DS per accedere a dati al di fuori di DS. Il DOS ed il BIOS utilizzano spesso ES come buffer per le loro chiamate.

Si sovrappongono ai registri a 32 bit (e.g. SS → [BP, DS] = EBP)

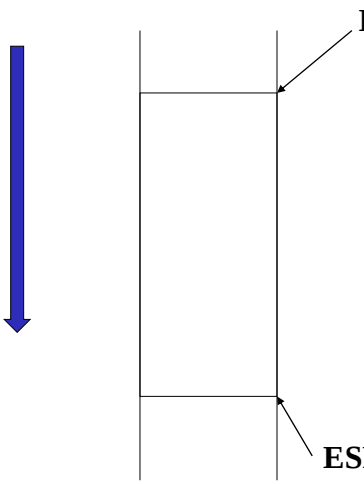
Utilizzo dei registri IA-32

- General Purpose Registers
  - General Data Registers
    - **EAX**: accumulatore (ottimizzato per op. aritmetico-logiche) – unico registro per gli operandi nell'architettura 8080 -> operandi in stack.
    - **EBX**: base register (registro base nel segmento dati, indirizzo corrente, $gp)
    - **ECX**: counter (ottimizzato per i cicli)
    - **EDX**: data register (contiene i dati, equivalente a uno dei registri $t)
  - General Address Registers
    - **EBP**: stack base pointer (base address dello stack, $fp)
    - **ESI**: source index (ottimizzato per op. su stringhe, e.g. movs)
    - **EDI**: destination index (ottimizzato per op. su stringhe, e.g. movs)
- Program counter e stack pointer
  - **EIP**: extended instruction pointer (EIP = Program Counter = SS || IP)
  - **ESP**: stack pointer ($sp) (SS || SP)
- Floating-point Stack registers
  - ST(0) – ST(7): 80 bit, accessibili come LIFO (stack, cf. registri $fp)
- Debug registers
  - DR(0) – DR(7): 32 bit, debugging della CPU.
- SIMD Registers
  - MMX, SSE (SSE1, SSE2, SSE3, SSE4, SSE5)
- Coordinamento tra: EBX e DS; tra EIP e SS; tra EBP, ESP e SS.

Utilizzo dei registri "General Data"

Nome simbolico			Nome descrittivo	Funzioni
32 bit	16 bit	8 bit		
EAX	AX	AH, AL	Accumulator	Moltiplicazione/Divisione, I/O, shift veloce
`out 70h, al` ; Il contenuto di al viene trasferito alla porta 70h. `sb $al,offset($k1)` ; In MIPS, accesso memoria > 2Gbyte				
EBX	BX	BH, BL	Base Register	Puntatore all'indirizzo base segmento dati
`mov edx, [ebx]` ;trasferisci quanto presente all'indirizzo 0($ebx) in edx `lw $s0, 0($gp)`				
ECX	CX	CH, CL	Count Register	Indice di conteggio per cicli, rotazioni e shift
move ecx, 10h ; load ecx con 10h (=16), valore di inizio conteggio (associato a "loop") **start: out 70h, al** ; Il contenuto di al viene trasferito alla porta 70h. **loop start** ; ritorna ad inizio ciclo, il quale verrà ripetuto 16 volte (fino a che ecx = 0)				
EDX	DX	DH, DL	Data Register	Moltiplicazione/Divisione, I/O
`mul edx` ; moltiplica edx con eax (implicito), il risultato è contenuto nella coppia edx:eax. ; in MIPS mult $t1, $t2 (risultato nella coppia di registri Hi:Lo)				

Lo stack viene delimitato



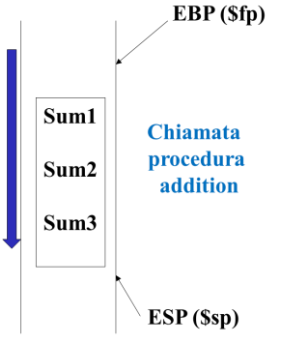
L'area di memoria privata della procedura (**frame di procedura**) è compresa tra EBP ed ESP. Viene utilizzato per memorizzare l'indirizzo di ritorno, per memorizzare i parametri di procedure e per memorizzare le variabili locali della procedura.

Non è obbligatorio utilizzare EBP. E' equivalente al ruolo di $fp in MIPS.

Registri di stack

Nome simbolico			Nome descrittivo	Funzioni
32 bit	16 bit	8 bit		
ESP	SP	x,x	Stack Pointer	Stack Pointer
EBP	BP,SS	x, x	Base Pointer	Indirizzo base del segmento di Stack (32 bit)

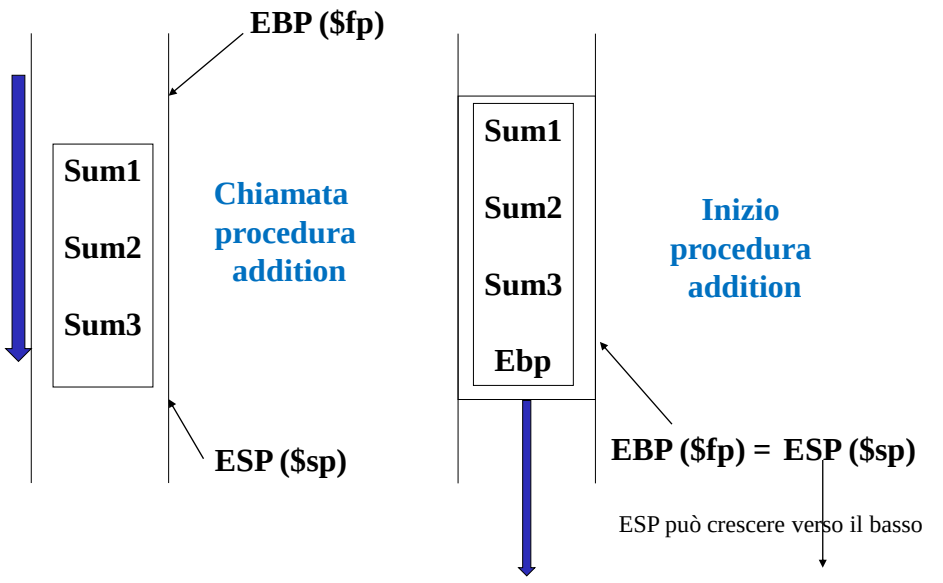
```
push sum1           ; create the first summand
                    ; addi $sp, $sp, -4; lw $s0, 0($sp)
                    ; (automaticamente ESP viene decrementato)
push sum2           ; create the second summand
push sum3           ; create the third summand
call addition       ; chiama procedura
                    ; addition (jal addition)
```



Situazione dello stack all'atto della chiamata
Il frame di procedura del chiamante

Il frame per addition inizierà da ESP e crescerà verso il basso

Lo stack viene delimitato

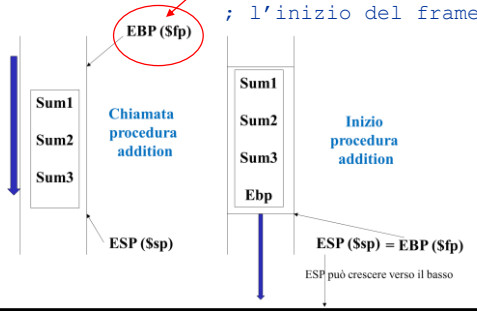


Registri di stack

Nome simbolico			Nome descrittivo	Funzioni
32 bit	16 bit	8 bit		
ESP	SP	x,x	Stack Pointer	Stack Pointer
EBP	BP,SS	x, x	Base Pointer	Indirizzo base del segmento di Stack (32 bit)

```
        call addition  ; chiama procedura addition (jal addition)

addition: proc near    ; call near (inside 64k segment, cf. branch)
   push ebp            ; salva l'indirizzo base per il ritorno
                       ; (inzio del record attivazione)
   move ebp, esp       ; copia lo StackPointer (esp) nel BaseP (individua
                       ; l'inizio del frame di procedura per addition)
```



Registri di stack

Architettura ad accumulatore

Nome simbolico			Nome descrittivo	Funzioni
32 bit	16 bit	8 bit		
ESP	SP	x,x	Stack Pointer	Stack Pointer
EBP	BP,SS	x, x	Base Pointer	Indirizzo base del segmento di Stack (32 bit)

```
   push sum1            ; create the first summand
                        ; addi $sp, $sp, -4; lw $s0, 0($sp)
                        ; (automaticamente ESP viene decrementato)
   push sum2            ; create the second summand
   push sum3            ; create the third summand
   call addition        ; chiama procedura addition (jal addition)

addition: proc near     ; call near (inside 64k segment, cf. branch)
   push ebp             ; salva l'indirizzo base per il ritorno
                        ;(inzio del record attivaz.)
   move ebp, esp        ; copia lo StackP (esp) nel BaseP (individua
                        ; frame di procedura)
   move eax, [ebp+12]   ; carica sum1 in EAX
   add eax, [ebp+8]     ; somma in eax sum1 + sum2
   add eax, [ebp+4]     ; somma in eax aum1 + sum2 + sum3
   pop ebp              ; recupera l'indirizzo base precedente
   ret                  ; ritorno al programma chiamante (cf. jr $ra)
addition endp
```

Indirizzi rispetto a EBP

IA-32 – operazioni logico-aritmetiche e confronto con il MIPS

Tipo operando 1	Tipo operando 2	Tipo risultato
Registro	Registro	Registro
Registro	Immediato	Registro
Registro	**Memoria**	**Registro**
Memoria	**Registro**	**Memoria**
Memoria	**Memoria**	**Memoria**

- Uno dei registri (memoria) deve fungere sia da operando che da registro destinazione (architettura ad accumulatore) E.g.: add eax, [ebp + 8] ➔ eax + [ebp + 8] ➔ eax.
  - MIPS permette di avere operandi e risultato in registri differenti
- Uno od entrambi gli operandi può provenire direttamente dalla memoria
  - Nel MIPS o nel PowerPC solo dai registri
- Gli operandi Immediati possono arrivare a 32 bit, gli altri ad 80 bit

Modalità di indirizzamento – istruzioni "immediate"

Modo	Descrizione	Codice MIPS	Codice INTEL
Indirizzamento immediato	L'operando è in una parte dell'istruzione	li $s0, 0x6a02	add eax,0x6a02
PC_relative addressing	Indirizzamento relativo al Program Counter (edx sottointeso)	bne $s0, $s1, label	sub eax jnz label
Psudodirect addressing	MIPS: indirizzo ottenuto cambiando i 26 bit dell'istruzione con i 28 LSB del PC (i 2 LSB sono 00) INTEL: modifica del registro Code Segment	j label	mov cs, 0x87ee mov ip 0x14fa = jump label = mov eip 0x87ee14fa

Esecuzione condizionata. Eseguo un'operazione in modo condizionato. Il risultato viene scritto se il flag (registro EFLAG) assume un certo valore. sub EAX, scrivere anche il flag di «Zero».

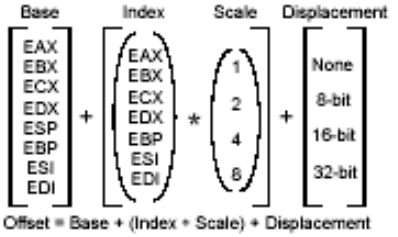
Non c'è più il limite dell'allineamento ai 32 bit della parola

Modalità di indirizzamento – dati

Modo	Descrizione	Restrizioni sui registri	Codice MIPS	Codice INTEL
Ind. diretto (registro) Register Addressing	Dato contenuto in un registro	No ESP e EBP	move $s0, $s1	mov eax, ebx
Indiretta tramite registro	Indirizzo del dato contenuto in un registro	No ESP e EBP	lw $s0, 0($s1)	mov eax, [ebx]
Base + offset (8 - 32bit)	Indirizzo = contenuto del registro base più offset	No ESP	lw $s0, 100($s1)	mov eax, 100 ([ebx] è sottinteso)

Base + indice scalato	Indirizzo = base + 2^{scale} * indice	Base: GPR qualsiasi
		Indice: no ESP
			mul $t0, $s2, 4
			add $t0, $t0, $s1
			lw $s0, 0($t0)
				mov eax, [esi*scala] ([ebx] è sottinteso)

Base + indice scalato + offset	Indirizzo = base + 2^{scale} *indice + offset	Base: GPR qualsiasi
		Indice: no ESP
			mul $t0, $s2, 4
			add $t0, $t0, $s1
			lw $s0, 24($t0)
				mov eax, [esi*scala + 24] ([ebx] è sottinteso)



Potenza espressiva. *Esempio, caricamento in EAX (sottointeso) dell'i-esimo elemento di un vettore:*

mov EBX, 0x0x224H ; indirizzo base
mov ESI, i
*mov EAX , [ESI*4]*

Interno ciclo

Dati di tipo diverso

- Possibilità di indirizzare dati da 8, 16, 32 (e 64 nell'estensione a 64 bit) bit.
- INTEL definisce un'ampiezza di default dei dati (16, 32 o 64 bit) .
- L'ampiezza di default viene impostata e modificata attraverso un **prefisso di 8 bit** inserito prima dell'istruzione (fino a 32 bit).

Ruolo del prefisso (1 byte) nell'8086:

- modificare i registri di segmento
- impedire l'accesso al bus
- ripetere l'istruzione (fino a quando ECX = 0, pensato per la move di stringhe o blocchi)
- dimensione di default dei dati e degli indirizzi.

Sotto-campo reg

Indica uno degli 8 registri in forma compatta (8 bit) o in forma estesa (16/32 bit).

reg	w = 0	w = 1	
		16b	32b
0	AL	AX	EAX
1	CL	CX	ECX
2	DL	DX	EDX
3	BL	BX	EBX
4	AH	SP	ESP
5	CH	BP	EBP
6	DH	SI	ESI
7	BH	DI	EDI

Codifica istruzioni – campi reg e w

- Campo **reg (cf. numero registro Register File in MIPS):**
  - la sua interpretazione dipende da w:
  - **w** determina il tipo di dato (short o long: 8 o 16/32 bit).
  - w=0 → registri a 8 bit
  - w=1 → 16 o 32 bit (dipende dal setting dei dati di default)
    - IA-32: 32 bit di default

→ istruzioni di lunghezza diversa

  - ADD AL, #1 → 16 bit
  - ADD EAX, #1 → 40 bit

4	3	1	8
ADD	000	0	0000 0001

4	3	1	32
ADD	000	1	00 … 01

d. PUSH ESI

5	3
PUSH	Reg

campo **reg**	**w=0**	**w=1**	
	8 bit	16 bit	32 bit
0	AL	AX	EAX
1	CL	CX	ECX
2	DL	DX	EDX
3	BL	BX	EBX
4	AH	SP	ESP
5	CH	BP	EBP
6	DH	SI	ESI
7	BH	DI	EDI

Specifica dell'indirizzamento dati

Pre-bit
w – specifica l'ampiezza (parte del codice operativo, 8, 16 o 32 bit)
reg – numero del registro.

Per alcune istruzioni il campo w e reg sono sufficienti.

Per altre istruzioni il codice operativo specifica direttamente anche il registro su cui si opera (e.g. ADD -> eax, ax) e la modalità di indirizzamento.

Per altre istruzioni esiste un post-byte (cf. campo function delle istruzioni di tipo R) per definire registry e modalità di indirizzamento. ***Per la modalità di indirizzamento con indice scalato esiste un secondo post-byte.***

Schema che consente soluzioni diverse, molto utilizzato nelle architetture Intel.

Post-byte 1:
- Mod (2bit) – modalità di indirizzamento
- r/m (3 bit) – variante della modalità di indirizzamento.

Codifica indirizzi – campi r/m e mod

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	*=(sib)*	SI+disp8	*(sib)*+disp8	SI+disp8	*(sib)*+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

- **r/m (3 bit)** seleziona il registro usato come **registro base** dell'indirizzo
- **mod (2 bit)** seleziona la **modalità di indirizzamento** (+ offset, offset+displacement,...)

mod = 3
Registro base dell'indirizzo = registro identificato dal campo reg.

Sotto-campo r/m (3 bit) e mod (2 bit)

r/m	mod = 0		mod = 1		mod = 2		mod = 3
	16b	32b	16b	32b	16b	32b	
0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	addr=SI	*=(sib)*	SI+disp8	*(sib)*+disp8	SI+disp8	*(sib)*+disp32	"
5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

r/m = 0, 1, 2, 3, 7
mod = 0 indica il registro base dell'indirizzo utilizzato di norma
mod = 1 coincide con mod = 0 con la differenza di un offset su 8 bit (disp8)
mod =2 coincide con mod = 0 con la differenza di un offset su 16 / 32 bit (disp16 / disp 32)

Registro a 32 bit per parole di 32 bit
Somma di due registri a 16 bit per parole di 16 bit (indirizzamento di segmento + offset)

sotto-campo r/m = 5, 6

r/m	mod = 0		mod = 1		mod = 2		mod = 3
	16b	32b	16b	32b	16b	32b	
0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	addr=SI	*=(sib)*	SI+disp8	*(sib)*+disp8	SI+disp8	*(sib)*+disp32	"
5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

r/m = 5– architettura a 16 bit
- **Come per r/m = 0, 1, 2, 3, 7**

r/m = 5– architettura a 32 bit
- **mod = 0** → Offset su 32 bit
- **mod=1,2** → Seleziona EBP + spiazzamento (8 / 32 bit)

r/m = 6 – architettura a 16 bit
- **mod = 0** → Offset su 16 bit
- **mod = 1,2** → indirizzo come BP + spiazzamento (8 / 16 bit)

r/m = 6 – architettura a 32 bit
- **Come per r/m = 0, 1, 2, 3, 7**

sotto-campo r/m = 4

r/m	mod = 0		mod = 1		mod = 2		mod = 3
	16b	32b	16b	32b	16b	32b	
0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	addr=SI	*=(sib)*	SI+disp8	*(sib)*+disp8	SI+disp8	*(sib)*+disp32	"
5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

architettura a 16 bit
- **Come per r/m = 0, 1, 2, 3, 7**

architettura a 32 bit – scaled index mode (sib)
- **mod = 0** → sib + 0
- **mod = 1** → sib + offset 8 bit
- **mod = 2** → sib + offset 32 bit

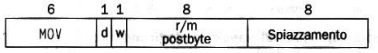
esi, scala, offset are specified in a second post-byte

Modo	Descrizione	Restrizioni sui registri	Codice MIPS	Codice INTEL

Base + indice scalato + offset	Indirizzo = base + 2^{scale} *indice + offset	Base: GPR qualsiasi
		Indice: no ESP
			mul $t0, $s2, 4
			add $t0, $t0, $s1
			lw $s0, 100($t0)
				mov eax, [esi*scala + 4]
				([ebx] è sottinteso)

Esempio

c. MOV EBX, [EDI + 45]

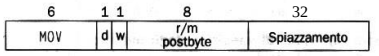


3 Byte

M[EDI + 45] -> EBX

d = 1 (lettura)
w = 1
r/m = 7
mod = 1
Spiazzamento = 45

c. MOV EBX, [EDI + 45]



6 Byte

M[EDI + 45] -> EBX

d = 1 (lettura)
w = 1
r/m = 7
mod = 2
Spiazzamento = 45

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	*=(sib)*	SI+disp8	*(sib)*+disp8	SI+disp8	*(sib)*+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

Sommario

Le architetture Intel
I registri ed il loro utilizzo
L'ISA degli Intel
La codifica delle istruzioni

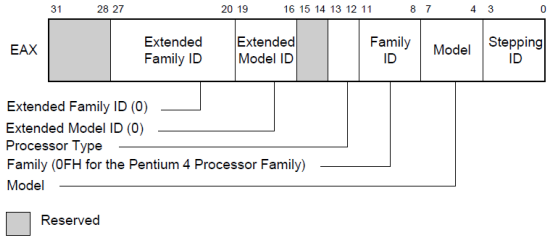
Quale CPU viene utilizzata?

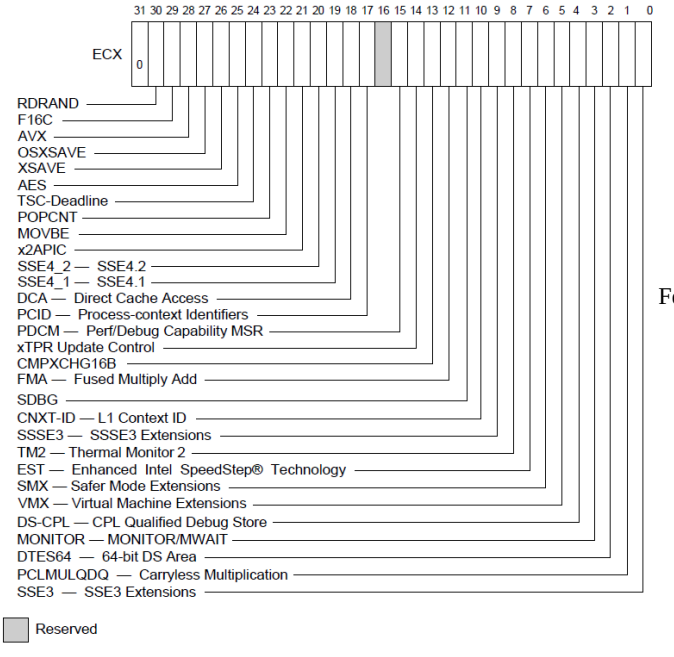
CPUID—CPU Identification

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

CPUID has no arguments. La richiesta viene codificata in EAX. Disponibile dal Pentium (1993). Prima chiamata con EAX = 0h, restituisce il tipo di CPU:

```
MOV EAX, 00H
CPUID
```





EAX = 01h

Features in **ECX** e EDX

Bit #	Mnemonic	Description
0	SSE3	**Streaming SIMD Extensions 3 (SSE3).** A value of 1 indicates the processor supports this technology.
1	PCLMULQDQ	A value of 1 indicates the processor supports PCLMULQDQ instruction.
2	DTES64	**64-bit DS Area.** A value of 1 indicates the processor supports DS area using 64-bit layout.
3	MONITOR	**MONITOR/MWAIT.** A value of 1 indicates the processor supports this feature.
4	DS-CPL	**CPL Qualified Debug Store.** A value of 1 indicates the processor supports the extensions to the Debug Store feature to allow for branch message storage qualified by CPL.
5	VMX	Virtual Machine Extensions. A value of 1 indicates that the processor supports this technology.
6	SMX	Safer Mode Extensions. A value of 1 indicates that the processor supports this technology. See Chapter 6, "Safer Mode Extensions Reference".
7	EST	**Enhanced Intel SpeedStep® Technology.** A value of 1 indicates that the processor supports this technology.
8	TM2	**Thermal Monitor 2.** A value of 1 indicates whether the processor supports this technology.
9	SSSE3	A value of 1 indicates the presence of the Supplemental Streaming SIMD Extensions 3 (SSSE3). A value of 0 indicates the instruction extensions are not present in the processor.
10	CNXT-ID	**L1 Context ID.** A value of 1 indicates the L1 data cache mode can be set to either adaptive mode or shared mode. A value of 0 indicates this feature is not supported. See definition of the IA32_MISC_ENABLE MSR Bit 24 (L1 Data Cache Context Mode) for details.
11	SDBG	A value of 1 indicates the processor supports IA32_DEBUG_INTERFACE MSR for silicon debug.
12	FMA	A value of 1 indicates the processor supports FMA extensions using YMM state.
13	CMPXCHG16B	**CMPXCHG16B Available.** A value of 1 indicates that the feature is available.
14	xTPR Update Control	**xTPR Update Control.** A value of 1 indicates that the processor supports changing IA32_MISC_ENABLES[bit 23].
15	PDCM	**Perfmon and Debug Capability.** A value of 1 indicates the processor supports the performance and debug feature indication MSR IA32_PERF_CAPABILITIES.
16	Reserved	Reserved
17	PCID	**Process-context identifiers.** A value of 1 indicates that the processor supports PCIDs and that software may set CR4.PCIDE to 1.
18	DCA	A value of 1 indicates the processor supports the ability to prefetch data from a memory mapped device.
19	SSE4.1	A value of 1 indicates that the processor supports SSE4.1.
20	SSE4.2	A value of 1 indicates that the processor supports SSE4.2.
21	x2APIC	A value of 1 indicates that the processor supports x2APIC feature.
22	MOVBE	A value of 1 indicates that the processor supports MOVBE instruction.
23	POPCNT	A value of 1 indicates that the processor supports the POPCNT instruction.
24	TSC-Deadline	A value of 1 indicates that the processor's local APIC timer supports one-shot operation using a TSC deadline value.
25	AES	A value of 1 indicates that the processor supports the AESNI instruction extensions.
26	XSAVE	A value of 1 indicates that the processor supports the XSAVE/XRSTOR processor extended states feature, the XSETBV/XGETBV instructions, and XCR0.
27	OSXSAVE	A value of 1 indicates that the OS has set CR4.OSXSAVE[bit 18] to enable XSETBV/XGETBV instructions to access XCR0 and to support processor extended state management using XSAVE/XRSTOR.
28	AVX	A value of 1 indicates that processor supports AVX instructions operating on 256-bit YMM state, and three-operand encoding of 256-bit and 128-bit SIMD instructions.

Features description

Quale CPU viene utilizzata?

CPUID—CPU Identification

Opcode	Instruction	64-Bit Mode	Compat/ Leg Mode	Description
0F A2	CPUID	Valid	Valid	Returns processor identification and feature information to the EAX, EBX, ECX, and EDX registers, as determined by input entered in EAX (in some cases, ECX as well).

Eempio: istruzioni vettoriali

La maggior parte delle istruzioni di base dell'estensione AVX-512 lavorano sui registri zmm a 512 bit + 8 registri di stato (opmask).

Per sapere se il processore supporta le istruzioni AVX-512 viene chiamata CPUID con **EAX = 07H, ECX = 0** e viene analizzato il bit 16 del registro EBX.

IA-32 Instruction Set

- Istruzioni general purpose
  - Istruzioni per lo spostamento dei dati
    - push, pop, utilizzo dello stack e della memoria dati
    - Tra registri
  - Istruzioni aritmetiche logiche, confronto e operazioni su interi e float
  - Istruzioni di controllo del flusso
    - basati sui flag (**condition codes**), allineamento al byte, salti condizionati e incondizionati, chiamata a procedura
  - Istruzioni della gestione delle stringhe (legacy mode)
  - Istruzioni di I/O

- Istruzioni floating point x87
  - funzioni trigonometriche, potenze di 2 (2^x, $\log_2 x$)

- Istruzioni SIMD
  - MMX, SSE (SSE2, SSE3, SSE4, SSE5), 3DNow! (IA-32 by AMD)

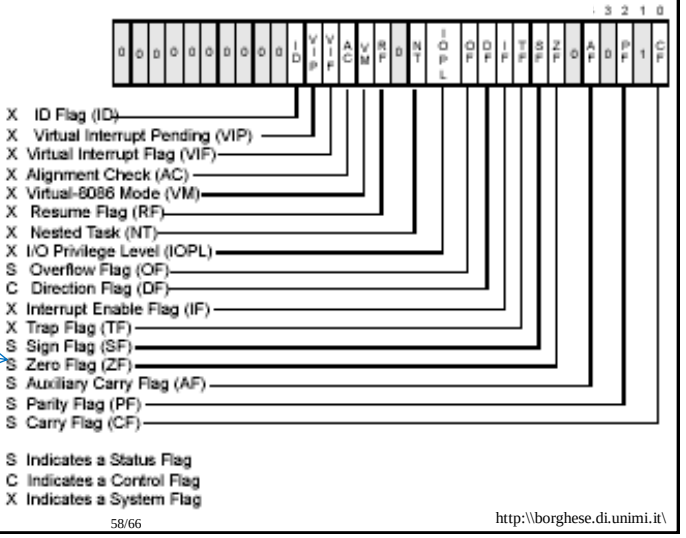
- Istruzioni di sistema
  - Cambio di modo, Halt, Reset, …

Registro EFLAG

- I risultati notevoli vengono salvati in questo registro
  - Carry, zero, overflow, segno, parità, …
  - Le istruzioni di branch si riferiscono sempre a EFLAG

MIPS:
beq $s0, $s1, label

IA-32:
sub eax
jz label



Alcuni esempi di IA-32 Instruction Set

Instruction	Meaning
Control	**Conditional and unconditional branches**
`jnz, jz`	Jump if condition to EIP + 8-bit offset; `JNE` (for `JNZ`), `JE` (for `JZ`) are alternative names
`jmp`	Unconditional jump—8-bit or 16-bit offset
`call`	Subroutine call—16-bit offset; return address pushed onto stack
`ret`	Pops return address from stack and jumps to it
`loop`	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX ≠ 0
Data transfer	**Move data between registers or between register and memory**
`move`	Move between two registers or between register and memory
`push, pop`	Push source operand on stack; pop operand from stack top to a register
`les`	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
`add, sub`	Add source to destination; subtract source from destination; register-memory format
`cmp`	Compare source and destination; register-memory format
`shl, shr, rcr`	Shift left; shift logical right; rotate right with carry condition code as fill
`cbw`	Convert byte in eight rightmost bits of EAX to 16-bit word in right of EAX
`test`	Logical AND of source and destination sets condition codes
`inc, dec`	Increment destination, decrement destination
`or, xor`	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
`movs`	Copies from string source to destination by incrementing ESI and EDI; may be repeated
`lods`	Loads a byte, word, or doubleword of a string into the EAX register

Descrizione di alcune istruzioni

JE, JZ: jump on equal - near (±128 byte) (MIPS **`beq…, beq $0,reg,…`**)

JMP: jump (near → uso CS; far → uso EIP) (MIPS **`j`**)

CALL: jump; SP=SP–4 (MIPS **`jal`**)

MOVW: lettura di una parola dalla memoria (MIPS **`lw`**)

PUSH,POP: aggiornamento implicito SP (MIPS **`addi $sp,$sp,±4)`**

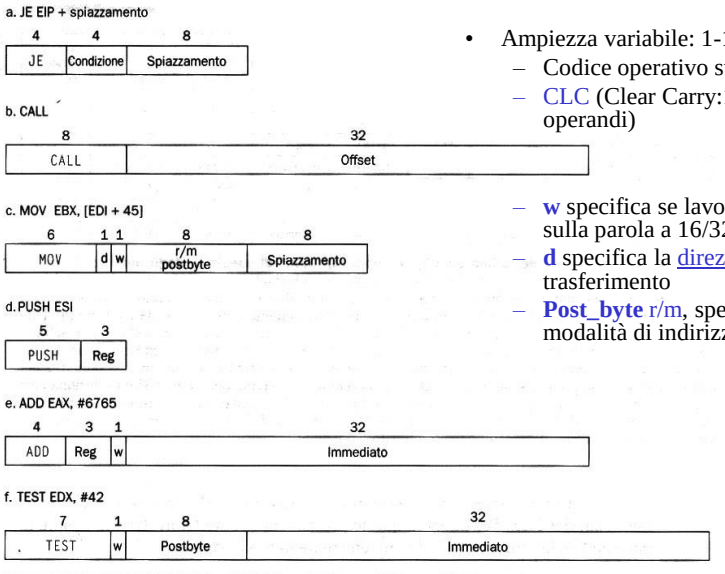
TEST: Carica nel flag il risultato di un'operazione (e,g $EDX AND 0x42)

MOVSL: Sposta 4 byte da memoria a memoria e incrementa EDI ed ESI (stringhe/aree dati)

Sommario

Le architetture Intel
Evoluzione degli Intel
L'ISA x86
La codifica delle istruzioni

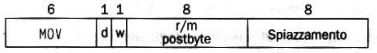
Codifica delle istruzioni



- Ampiezza variabile: 1-15 byte.
  - Codice operativo su 1 o 2 byte.
  - CLC (Clear Carry:1 byte, non ha operandi)
  - **w** specifica se lavora sul byte o sulla parola a 16/32 bit (word)
  - **d** specifica la direzione del trasferimento
  - **Post_byte** r/m, specifica la modalità di indirizzamento

Esempio

c. MOV EBX, [EDI + 45]

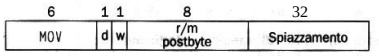


3 Byte

M[EDI + 45] -> EBX

d = 1 (lettura)
w = 1
r/m = 7
mod = 1
Spiazzamento = 45

c. MOV EBX, [EDI + 45]



6 Byte

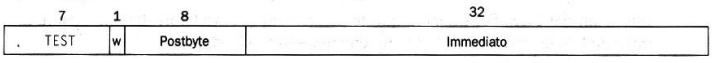
M[EDI + 45] -> EBX

d = 1 (lettura)
w = 1
r/m = 7
mod = 2
Spiazzamento = 45

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	*=(sib)*	SI+disp8	*(sib)*+disp8	SI+disp8	*(sib)*+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

Esempio

f. TEST EDX, #42



If (EDX == 42) Flag == 1

w = 1
r/m = 5
mod = 0
Immediato = 42

reg	w = 0	w = 1		r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr as*	*addr as*	*addr as*	*addr as*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+ disp8*	*+ disp8*	*+ disp16*	*+ disp32*	*field*
4	AH	SP	ESP	4	addr=SI	*=(sib)*	SI+disp8	*(sib)*+disp8	SI+disp8	*(sib)*+disp32	"
5	CH	BP	EBP	5	addr=DI	=disp32	DI+disp8	EBP+disp8	DI+disp16	EBP+disp32	"
6	DH	SI	ESI	6	addr=disp16	=ESI	BP+disp8	ESI+disp8	BP+disp16	ESI+disp32	"
7	BH	DI	EDI	7	addr=BX	=EDI	BX+disp8	EDI+disp8	BX+disp16	EDI+disp32	"

Codifica istruzioni: osservazioni

- Architettura CISC
  - Lunghezza variabile istruzioni
  - Lunghezza variabile dei campi del Codice Operativo (primo Byte)
- La lunghezza dell'istruzione dipende dal contenuto di alcuni campi dell'istruzione stessa
  - mod, r/m, reg, w
  - Devo iniziare la decodifica dell'istruzione per sapere quant'è lunga → prima di terminare la fase di fetch
- Architettura complessa e poco razionale
  - prezzo da pagare per mantenere la compatibilità verso il basso (8, 16, 32 bit)
- **Perdiodicamente Intel considera di virtualizzare le architetture vintage**
- **Elenco delle istruzioni Intel** (alcune vengono mantenute solo per compatibilità): https://en.wikipedia.org/wiki/X86_instruction_listings

Sommario

Le architetture Intel
Evoluzione degli Intel
L'ISA x86
La codifica delle istruzioni