Calcolatori Elettronici L- A

Architetture dei Microprocessori

Architetture...

Durante il corso si trattano due diverse “architetture”:

- Architettura **dei microprocessori** (e più in generale le loro ISA)
- Architettura **dei calcolatori** o sistemi a μP (quindi progettazione di sistemi con interfacciamento dei processori alle periferiche seriali, hard-disk, ecc.)

L’architettura dei calcolatori (o dei sistemi a microprocessore) sarà affrontata nella parte finale del corso.

Adesso esamineremo le architetture dei microprocessori.

Questo termine fu coniato da IBM nel 1964 per la CPU IBM 360. Amdhal, Blaauw e Brooks usarono questo termine per riferirsi alla parte dell’insieme di istruzioni visibile al programmatore. Essi infatti ritenevano che una famiglia di macchine della stessa architettura dovesse essere in grado di eseguire lo stesso software. Il concetto di una industria standardizzata su di una singola architettura fu un’innovazione radicale e di successo, tanto che ancora oggi i maggiori produttori producono CPU diverse ma basate sulla stessa “architettura” (intel e AMD accettano lo stesso SW), quindi compatibili... anzi “IBM compatibili” (da Hennessy Patterson par. 3.11)

Sommario

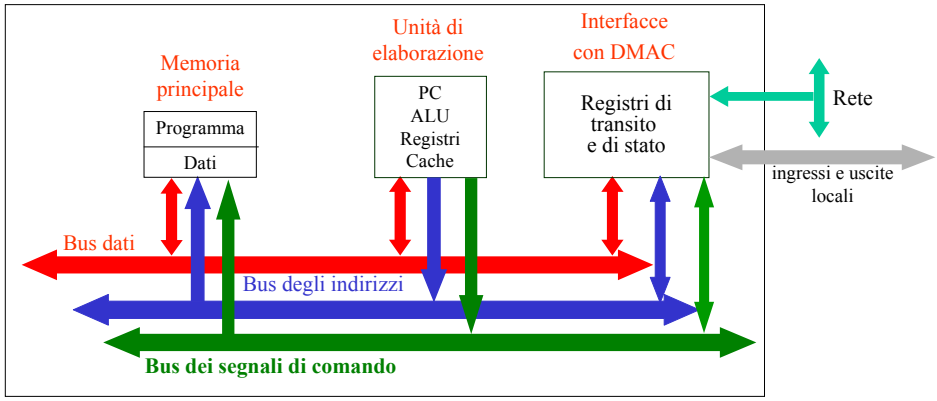
- Richiami
- Definizione di Instruction Set Architecture - ISA
- Classificazione delle ISA
- Notazione RTL
- ISA del processore DLX
- ISA iA16 dei processori 8086-8088
- ISA iA32 dei processori Pentium
- Misure sulla frequenza di esecuzione delle istruzioni al variare dell'ISA

Richiamo:
architettura di un sistema a µP

- L'architettura del calcolatore è una terna i cui componenti sono:
  - Il linguaggio macchina (detto anche set di istruzioni o istruction set)
  - La struttura interna (costituita da un insieme di blocchi interconnessi) il cui progetto teorico trova un corrispondente hardware nella
  - realizzazione circuitale (cioè la tecnologia microelettronica impiegata per la realizzazione)

Richiamo: rappresentazione visuale della struttura dell'hardware di un calcolatore

• *L'hardware del calcolatore si interfaccia con il software attraverso il suo set di istruzioni (linguaggio macchina)*

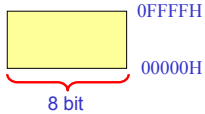


Richiamo: spazio di indirizzamento in memoria

- Gli indirizzi si indicano solitamente in codice esadecimale
- la dimensione di uno spazio di indirizzamento si può esprimere in:
  - Kilobyte (1KB = 2^{10} Byte = 1024 B)
  - Megabyte (1MB = 1K KB = 2^{20} Byte = 1.048.576 B)
  - Gigabyte (1GB = 1K MB = 2^{30} Byte = 1.0e+9)
- Esempi
  - il Pentium e il DLX hanno uno spazio di indirizzamento di 4 GB (indirizzi di 32 bit rappresentabili con 8 cifre esadecimali)
  - l'8086 ha uno uno spazio di indirizzamento di 1 MB (indirizzi di 20 bit rappresentabili con 5 cifre esadecimali)

Richiamo:
Interfacce di ingresso/uscita
Spazio di indirizzamento in I/O

- Le interfacce di I/O contengono registri su cui transitano i dati scambiati con il mondo esterno
- Così come i dispositivi di memoria, anche le interfacce di ingresso/uscita (e precisamente i registri di transito dei dati e ogni altro registro indirizzabile dal software) vanno mappate in uno spazio di indirizzamento
- Le interfacce di I/O possono essere mappate in uno spazio distinto da quello della memoria oppure nello stesso; in quest'ultimo caso si dice che l'I/O è mappato in memoria (***memory mapped I/O***)
- Lo spazio di indirizzamento in I/O è solitamente più piccolo dello spazio di indirizzamento in memoria; es: nelle architetture Intel iA16 e iA32 (intel Architecture con bus dati a 16bit e 32bit) lo spazio di indirizzamento in I/O è di 64 KB

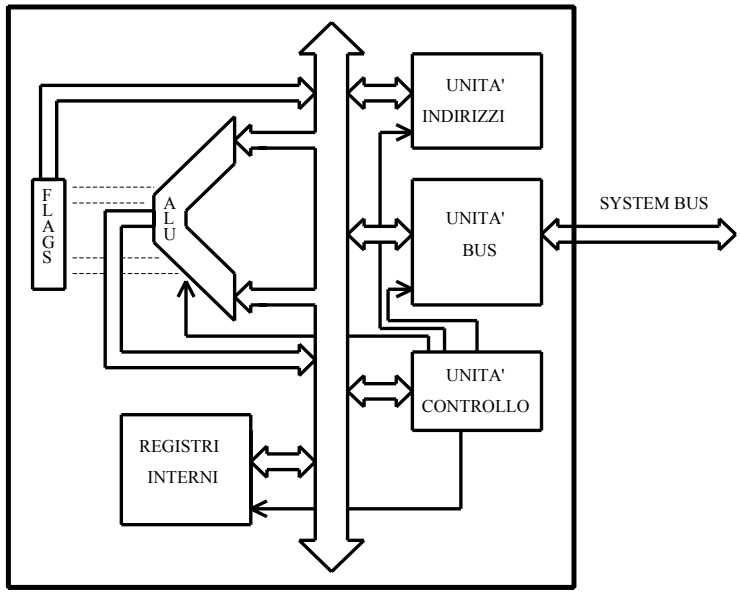


Spazio di indirizzamento in I/O di 64 KB
Es. iA16 e iA32

Definizione di Linguaggio Macchina (Instruction Set Architecture)

- Il linguaggio macchina (L.M.) (detto anche ISA “Instruction Set Architecture”) è il livello dell’architettura della CPU visibile a chi sviluppa i compilatori e a chi programma in assembler
- L’ISA è costituita dall’insieme delle istruzioni eseguibili e dal loro formato binario
- Con le istruzioni del linguaggio macchina si accede alle risorse interne del calcolatore:
  - memoria
  - registri
  - flag (sono le variabili di stato, dette anche variabili di sistema)
- Tra le risorse interne al calcolatore, solo quelle accessibili attraverso l’ISA possono essere rese visibili e controllabili dal software

Architettura interna di un µP



Principale requisito di un linguaggio macchina

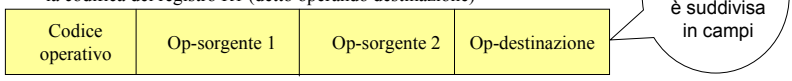
- Un Linguaggio macchina (ISA) viene principalmente valutato in base all'efficienza del codice generato dai compilatori
- Il progettista di un'ISA deve consentire la concezione di compilatori capaci di generare codice che minimizzi il valore di CPU_{time} per il maggior numero di applicazioni possibili

$$(CPU_{time} = N_{istruzioni} * CPI_{medio} * T_{ck})$$

- L'evoluzione delle applicazioni influenza i L.M. (es. le applicazioni multimediali hanno portato a estensioni del L.M. su diverse architetture)

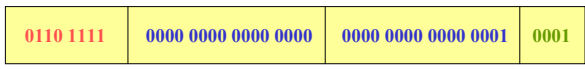
Codifica binaria delle istruzioni in L.M.

- Esempio: supponiamo che nel L.M. della nostra **ipotetica CPU** esista una istruzione che sommi tra di loro due variabili (es. A e B) e ponga il risultato in un registro interno alla CPU (es. R1). L'istruzione in linguaggio macchina dovrà contenere in forma binaria le seguenti informazioni:
  - la codifica dell'operazione somma (detta codice operativo o Opcode)
  - la codifica degli indirizzi di A e B (detti operandi sorgente)
  - la codifica del registro R1 (detto operando destinazione)



L'istruzione macchina è suddivisa in campi

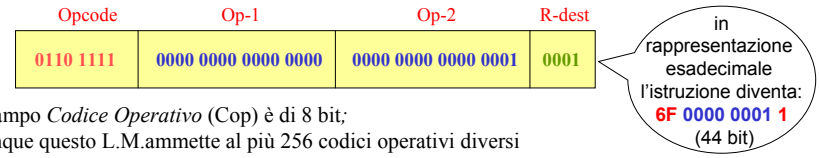
Questo è un esempio di formato di istruzione in L.M. per una **ipotetica CPU**
Ogni campo è una stringa di bit.
L'istruzione di somma proposta potrebbe essere codificata come segue:



in rappresentazione esadecimale l'istruzione diventa: **6F 0000 0001 1** (44 bit)

Le istruzioni del L.M. per essere interpretate dalla CPU devono essere codificate in forma binaria. Per comodità noi le rappresenteremo in codice esadecimale.

Considerazioni sull'esempio di formato di istruzione appena visto



- Il campo *Codice Operativo* (Cop) è di 8 bit*;*
  dunque questo L.M.ammette al più 256 codici operativi diversi
- Nel codice operativo sono implicite le seguenti informazioni:
  - l'operazione è una operazione di somma
  - gli operandi sorgente sono in memoria e il loro indirizzo è codificato nel secondo e nel terzo campo
  - l'operando destinazione è un registro interno identificato dal quarto campo
- I campi Op-1 e Op-2 (indirizzi degli operandi sorgente) sono di 16 bit, dunque l'istruzione può indirizzare fino a 2^{16} = 64 K celle di memoria distinte
- Le variabili da sommare (Op-1 e Op-2) si trovano agli indirizzi 0 e 1 della memoria
- Il registro su cui memorizzare il risultato è il registro individuato dal numero binario di 4 bit 0001B. Dunque con questo Codice Operativo si possono indirizzare al massimo 16 registri interni (probabilmente la CPU dispone di soli 16 registri di uso generale disponibili al programmatore)

Rappresentazione simbolica delle istruzioni macchina: l'assembler

- Per comodità in generale non rappresenteremo le istruzioni macchina con zeri e uni, né con cifre esadecimali ma **in forma simbolica**; il linguaggio che useremo per rappresentare simbolicamente le istruzioni del linguaggio macchina è l'assembler
- Il formato tipico di una istruzione in assembler è il seguente:

etichetta	codice mnemonico	lista operandi

separati da virgola (destinazione al primo posto: D,S,S...)

- L'istruzione del lucido precedente può essere rappresentata in assembler come segue:

Linguaggio macchina	**6F 0000 0001 1**
Linguaggio assembler	**add r1, A, B**

  - **add** è il codice mnemonico associato al codice operativo **6F** (operazione di "somma" per la CPU)
  - 0 è l'indirizzo in memoria di A e 1 è l'indirizzo di B
  - **r1** è il simbolo associato al **registro 1**
  - dunque: abbiamo adottato la convenzione di indicare in minuscolo gli identificatori specifici dell'architettura (registro interno e cop), e di indicare in maiuscolo le variabili in memoria
  - l'etichetta è facoltativa e serve per rappresentare in modo simbolico l'indirizzo dell'istruzione, cioè l'indirizzo ove l'istruzione è memorizzata (ad es. nei loop può essere "next", "loop", "ciclo"...)

Classificazione dei Linguaggi Macchina

- Proprietà dei LM che esamineremo e che permettono di classificarli:
  - insieme delle operazioni eseguibili
  - tecniche di indirizzamento (cioe' modalita' di calcolo dell'indirizzo degli operandi in memoria)
  - sede degli operandi all'interno della CPU (stack, accumulatore, registri)
  - numero di operandi definiti esplicitamente nel formato dell'istruzione (0/1/2/3 operandi)
  - numero degli indirizzi di memoria referenziati in una istruzione (0/1/2/3 indirizzi)
  - presenza di operandi nel formato della istruzione (operandi immediati)
  - tipologia, dimensione e organizzazione degli operandi
  - lunghezza dell'istruzione (fissa o variabile)

Ogni µP ha un suo assembler con una propria sintassi ma, tra loro, i vari assembler si assomigliano.

LM: tipologia delle istruzioni

Le istruzioni di un linguaggio macchina si possono suddividere nei seguenti sottoinsiemi:

- istruzioni aritmetiche e logiche (Es. ADD, AND, INC, DIV, etc.)
- istruzioni di trasferimento dati (Es. MOV, LD, ST, PUSH, IN,OUT)
- istruzioni di trasferimento del controllo (Es. JMP, CALL, RET, JCC)
- istruzioni di sistema (Es. STI, CPUID,...)
- istruzioni che operano su operandi in virgola mobile (Es. FLD, FMUL, FSQRT)

L'insieme completo delle istruzioni assembler di un µP è detto Instruction Set

LM: tecniche di indirizzamento degli operandi in memoria

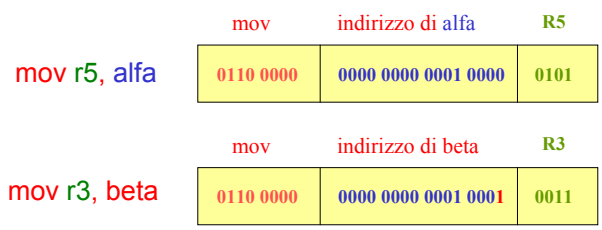
- Due obiettivi:
  - 1) consentire l'accesso a indirizzi **definiti staticamente** (cioè noti al momento del caricamento del programma "load time")
  - 2) consentire l'accesso a indirizzi **calcolati dinamicamente** (cioè calcolati a "run time"):
- Nel primo caso (indirizzamento diretto o indirizzamento per nome) l'indirizzo dell'operando compare nell'istruzione in L.M.
  - esempio ISA iA16[(1)]: mov ax, alfa; l'indirizzo dell'operando in memoria deve comparire esplicitamente nell'istruzione in forma binaria
- Nel secondo caso (indirizzamento indiretto) l'istruzione deve contenere le informazioni necessarie a reperire i componenti dell'indirizzo e a calcolarlo
  - es. ISA iA16: mov ax, alfa[SI]; SI è un registro che contiene l'indice del vettore; l'indirizzo dell'operando è la somma di alfa (costante) + il contenuto di SI, registro il cui valore varia durante l'esecuzione del programma (a run time)

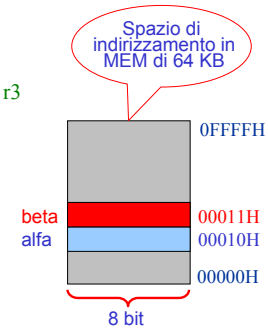
[(1)] (ISA dell'intel Architecture con bus dati a 16bit)

LM: indirizzamento diretto o “per nome”

L’indirizzo dell’operando compare esplicitamente nell’istruzione in L.M.

Esempio per una ipotetica CPU:
Supponiamo che alfa e beta siano due variabili assegnate agli indirizzi 10H e 11H dello spazio di indirizzamento in memoria. Vediamo due istruzioni in LM che copiano i valori di alfa e beta sui due registri r5 ed r3

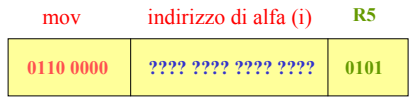




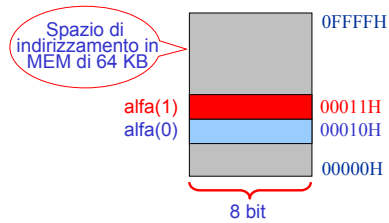
- l’indirizzo dell’operando in memoria (alfa e beta nei due esempi) compare esplicitamente nel codice dell’istruzione

LM: accesso a indirizzi che cambiano a run time (ind. indiretto)

- La tecnica di indirizzamento del lucido precedente non consente di accedere con la stessa istruzione a indirizzi che cambiano durante l'esecuzione (es. accesso a un array attraverso l'indice modificato a run time)
- Es.: se alfa(0..1) è un vettore di due byte posizionato all'indirizzo 10H, allora:
- l'indirizzo di alfa(0) è 10H; l'istruzione mov r5,alfa(0) è 60 0010 5
- l'indirizzo di alfa(1) è 11H; l'istruzione mov r5,alfa(1) è 60 0011 5
- se volessi eseguire l'istruzione: mov r5, alfa(i) con i variabile a run time, dovrei modificare il codice eseguibile; ma il codice eseguibile non può essere modificato, quindi concludiamo che per indirizzare gli elementi di un vettore è necessario un nuovo formato di istruzione



Mov r5, alfa(i)



LM: esempio di utilizzo dell'indirizzamento indiretto

- Supponiamo di voler calcolare la somma degli elementi di un vettore ALFA[0..9] di 10 elementi:

$$S = \Sigma \text{ ALFA(i)}$$

- Il calcolo può essere effettuato con un loop di 10 iterazioni, appoggiandosi a un indice di loop i e a un registro temporaneo r5:

```
        mov     i,0
        mov     r5,0
ciclo   add     r5, ALFA (i)
        add     i,1
        if  i < 10 than go to ciclo
        mov     S, r5
```

- la terza istruzione accede a un indirizzo il cui valore cambia durante l'esecuzione
- non potendo modificare a run time l'istruzione in L.M. per aggiornarne l'operando (il codice non è modificabile!), dobbiamo definire un'istruzione in cui sia possibile cambiare dinamicamente l'indirizzo dell'operando
- l'istruzione dovrà contenere le informazioni necessarie a reperire i componenti dell'indirizzo e a calcolarlo
- l'istruzione dovrà effettuare due calcoli: indirizzo e risultato!

LM: il modo più semplice per introdurre l'indirizzamento indiretto nel formato delle istruzioni

INDIRIZZAMENTO TRAMITE REGISTRO E OFFSET

- mov r1, alfa (r2)
- in caso di accesso a un vettore, il registro r2 funge da indice e l'offset alfa è la base
- l'indirizzo dell'operando si ottiene sommando ad alfa (costante) il contenuto del registro r2 (variabile a run time)
- se r2 = 0 (costante), allora si ricade nel caso dell'indirizzamento diretto

- Ad esempio il formato di una istruzione con questa modalità di indirizzamento può essere il seguente:

Opcode	R-dest	R-indice	offset
0000 1111	0001	0010	0000 0000 0001 0000

- Con questo formato di istruzione è possibile codificare sia l'indirizzamento diretto sia l'indirizzamento indiretto; è conveniente, a tal fine, disporre di un registro che contenga la costante 0 (in diverse architetture il registro r0 contiene la costate 0)

Il concetto di Stack

• Area di memoria gestita LIFO (ad esempio, può contenere i record di attivazione delle procedure)

• realizzato in memoria centrale

• definito da una coppia di registri uno dei quali (Base) indica l'indirizzo di partenza dell'area e l'altro (Stack Pointer) la prima locazione libera sopra la cima dello stack (Top)

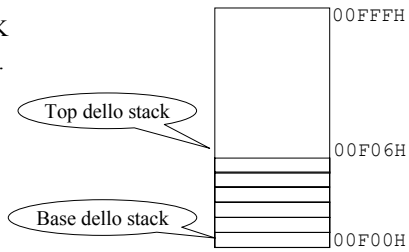
• Talune CPU consentono la coesistenza in memoria di più stack, ma spesso si limita la possibilità di usarne uno alla volta (un solo stack attivo per volta)

Esempio di STACK da 256 locazioni con Base = 0H e Stack Pointer = 6H

ISTRUZIONI DI ACCESSO ALLO STACK

L'istruzione PUSH aggiunge un dato all'indir. puntato da TOP e lo incrementa

L'istruzione POP legge il dato puntato da (TOP-1) e lo toglie allo stack decrementando TOP (rendendo così cancellabile il contenuto del nuovo TOP)



Sede degli operandi all'interno della CPU

- La **sede** in cui sono memorizzati gli operandi di una una ***tipica istruzione ALU*** costituisce una delle caratteristiche fondamentali di un ISA. Esistono 3 approcci principali:
- **Stack Architecture**: sia i due operandi sorgente sia l'operando destinazione sono in cima allo stack.
- **Accumulator Architecture**: Uno dei due operandi sorgente è un registro speciale, detto ***accumulatore***. L'accumulatore contiene anche il risultato dell'operazione.
- **General-Purpose Register (GPR) Architecture**: gli operandi risiedono in registri interni oppure in memoria.

Esempio: C = A + B

STACK	ACCUMULATOR	GPR
PUSH A	LOAD A	LOAD R1,A
PUSH B	ADD B	ADD R1,B
ADD	STORE C	STORE C,R1
POP C		

Numero di operandi presenti in modo esplicito nel formato delle istruzioni che utilizzano l'ALU

- Gli operandi di una istruzione aritmetica o logica sono 3:
  - 2 operandi sorgente e un operando destinazione
- Uno dei criteri in base ai quali si caratterizza un'ISA è il numero di operandi che compaiono esplicitamente nel formato delle istruzioni ALU, cioè per i quali nel formato della istruzione c'è un campo dedicato.
- In particolare il numero di operandi può essere:
  - 3 (es. add A, B, C)
  - 2 (es. add A, r1)
  - 1 (es. add A)
  - 0 (es. add)
- gli operandi che non compaiono esplicitamente nel formato dell'istruzione sono definiti in modo implicito; es.:
  - add A, r1 significa: A ← A + r1 la destinazione coincide con uno dei due op. sorgente
  - add presuppone, ad esempio una architettura a stack; è implicito che gli operandi siano nello stack; i due elementi in cima allo stack vengono sostituiti dal risultato

Numero di indirizzi di memoria presenti nel formato delle istruzioni che utilizzano l'ALU

- Gli operandi di una istruzione aritmetica o logica possono risiedere in memoria o in un registro interno alla CPU
- Uno dei criteri in base ai quali si caratterizza un'ISA è il numero di diversi indirizzi in memoria che compaiono esplicitamente nel formato delle istruzioni ALU, cioè per i quali nel formato della istruzione ci sono i campi necessari a calcolarlo.
- In particolare questo numero può essere:
  - 3 (es. add A, B, C) — A, B, C sono tre distinte variabili in memoria, questa istruzione si chiama istruzione a tre indirizzi
  - 2 (es. add A, B) — A e B sono due distinte variabili in memoria, questa istruzione si chiama istruzione a due indirizzi
  - 1 (es. add A,r1) — A è una variabile in memoria e r1 è un registro, questa istruzione si chiama istruzione a un indirizzo
  - 0 (es. add r3,r2,r1) — gli operandi sono tutti registri interni alla CPU, questa istruzione si chiama istruzione a zero indirizzi

Le 3 classi più importanti di ISA GPR

- Consideriamo le architetture GPR: le possibili combinazioni di (num. operandi - num. indirizzi) di una tipica istruzione ALU sono 7:

  (3-0), (3-1), (3-2), (3-3), (2-0), (2-1), (2-2)

  Tuttavia, le 3 classi
  (3,0) → R-R (Register-Register)
  (2,1) → M-R (Memory -Register)
  (3,3) → M-M (Memory - Memory)
  sono sufficienti per classificare quasi tutte le architetture GPR esistenti.

- Alcuni esempi
  R-R: macchine RISC; M-R: iA; M-M: VAX

Tipologia, dimensione e organizzazione degli operandi

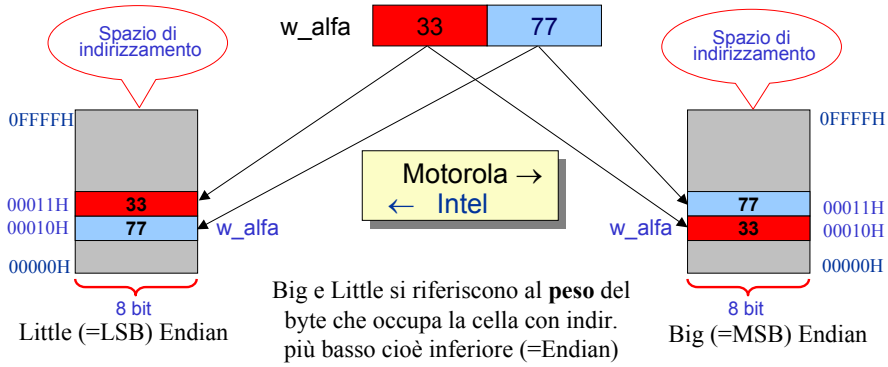
- Gli operandi di una istruzione aritmetica o logica possono essere:
  - byte
  - quantità di 16 bit (dette WORD in iA e HALF-WORD nelle architetture R-R)
  - quantità di 32 bit (dette DOUBLE WORD in iA e WORD nelle architetture R-R)
  - quantità di 64 bit (dette Q-WORD in iA e D-WORD nelle architetture R-R)
- L'interpretazione degli operandi (numeri interi con segno, senza segno, reali, etc.) è solitamente affidata all'istruzione, dunque è fissata dal codice operativo

Nota: il termine "word" è nato per esprimere la lunghezza della "parola parlata" da un µP, quindi ogni processore aveva una sua word, definita univocamente in funzione della dimensione del bus dati. Tuttavia in seguito "word" ha assunto il significato di lunghezze fisse (il più comune è word = 16bit) definito arbitrariamente dai vari progettisti e case costruttrici.

Perciò "word" va sempre intesa criticamente in quanto, ad esempio, la stessa casa produttrice di due µP a 16 e 32bit potrebbe sempre intendere word = 16bit; viceversa un'altra casa con lo stesso tipo di prodotti a catalogo potrebbe intendere per entrambi word = 32bit

Come vengono memorizzati i dati “multibyte” in memoria? Convenzioni Little e Big Endian

- Si consideri una variabile di due byte il cui contenuto sia ad esempio 03377H lunga cioè 2 byte.
- Supponiamo che w_alfa si trovi in memoria centrale a partire dall’indirizzo 10H; dunque w_alfa occupa le due celle di memoria di indirizzo 10H e 11H.
- W_alfa può essere memorizzato in uno dei due seguenti modi:



Big e Little si riferiscono al **peso** del byte che occupa la cella con indir. più basso cioè inferiore (=Endian)

Allineamento dei dati

- Si consideri un blocco *B* di dati di *m* byte, con *m* una potenza del 2 (*m=2**k*)
- Supponiamo che *B* sia memorizzato in memoria all'indirizzo *A*
- Si dice che *B* è allineato se *A* è un multiplo di *m*, cioè se:
  - **indirizzo (B) $_{\text{mod m}}$ = 0** (condizione di allineamento)
- Esempi:
  - Un dato di due byte (k=1) è allineato se il suo indirizzo è pari, cioè ha LSbit=0
  - Una dato di 8 byte (k=3) è allineato se il suo indirizzo binario termina con 3 zeri
  - Un blocco di 16 byte (k=4, detto *paragrafo*) è allineato se il suo indirizzo in codice esadecimale ha la cifra meno significativa uguale a zero
  - Un blocco di 4 KB (k=12, detto anche *pagina di 4 KB*) è allineato se il suo indirizzo in codice esadecimale ha le tre cifre meno significative uguali a zero
- Alcune ISA (R-R) impongono l'allineamento dei dati allo scopo di rendere più rapidi lo scambio e l'elaborazione dei dati (iA non lo richiede, ma è più veloce a trattare dati allineati)

Tabella comparativa dei linguaggi macchina per architetture a 32 bit considerati nel Corso

	Denominaz. architettura	*Formato e lunghezza istruzioni*	*Sede operandi nella CPU*	*Numero operandi espliciti in istruz. ALU*	*Numero max operandi in memoria in istruz. ALU*	*Numero registri visibili nelle applicazioni*	*Esempi*
DLX (3-0)	• RISC • Registro/ registro • Load/Store • 3-0 • R-R	• Lunghezza costante • 32 bit • Formati: R, I, J	• Registri	• 3 • Es.: ADD R5, R6, R3	• 0 (macchine a zero indirizzi) • Es: ADD R1, R2, 5	• 32 General purpose da 32 bit • 32 Floating point da 32 bit	• DLX • MIPS • SPARC • Power PC • HP-PA • I860
iA (2-1)	• CISC • Memoria/ registro • 2-1 • M-R	• Lunghezza variabile (multipla di un byte)	• Registri	• 2 • Es: ADD AX,BX	• 1 (macchine a un indirizzo) • Es.: ADD alfa, AX	• 8 General purpose da 32 bit • 8 Floating point da 80 bit (stack) • 6 Segment registers	• Intel a 32 bit • Motorola 68000 • IBM 360

Ulteriore differenza tra i L.M. delle architetture R-R e M-R sta nelle modalità di indirizzamento degli operandi in memoria

ISA di tipo RISC e CISC a confronto

1) CISC: Complex Instruction Set Computing
2) RISC: Reduced " " "

1) hanno un ISA con istruzioni ASM complesse simili a un linguaggio di alto livello; con poche istruzioni si scrive codice potente; le istruzioni possono elaborare operandi da memoria; ampia scelta delle modalità di indirizzamento; elevato CPI

2) hanno un ISA con poche elementari istruzioni; servono molte istruzioni per scrivere semplici programmi; le istruzioni non possono elaborare direttamente operandi da memoria; pochissime modalità di indirizzamento; CPI basso

$$CPU_{time} = N_{istruzioni} * CPI_{medio} * T_{ck}$$

A parità di T_{ck} e di tecnologia è stato dimostrato che le architetture RISC sono più veloci perché la riduzione del CPI_{medio} supera l'incremento di istruzioni necessarie a scrivere programmi.

Ma un fattore altrettanto importante per il confronto delle prestazioni dei μP è l'introduzione della tecnica di Pipelining...

ISA considerate nel Corso

- In questo corso studieremo le seguenti architetture:
  - 1- architetture a 0 indirizzi con tre operandi espliciti, dette anche "**architetture 3-0**", architetture "Load/Store" o "**Registro/Registro**" o **RISC**; hanno solitamente istruzioni di lunghezza costante pari a 32 bit. Esempi sono: il DLX, il Power PC, il MIPS, la SPARC, l'architettura HP PA, l'i860
  - 2 - architetture a un indirizzo con due operandi espliciti per istruzione dette anche "**architetture 2-1**" o architetture "**memoria-registro**" (hanno solitamente istruzioni di lunghezza variabile, es. iA - Intel Architecture)

Calcolatori Elettronici L- A

La notazione RTL

Notazione RTL (1)

Notazione RTL

- I seguenti simboli consentono di costruire in modo estremamente efficiente configurazioni binarie
- Traslazione logica a sinistra di n bit: << n
- Traslazione logica a destra di n bit: >> n
- Concatenazione di due campi: ##
- Ripetizione di x n volte: $(x)^n$
- Ennesimo bit di una conf. binaria x: x_n (il pedice seleziona un bit)
- Selezione di un campo in una stringa di bit x: $x_{n..m}$ (un range in pedice seleziona il campo)

- Es.: data la configurazione binaria di 8 bit C= 01101100B, si ha:
  - C << 2 : 10110000B
  - $C_{3..0}$ ## 1111: 11001111B
  - $(C_{3..0})^2$:11001100
  - $(C_6)^4$ ## C>>4: 111100000110

Notazione RTL (2)

- Trasferimento di un dato: $\leftarrow$
  Il numero di bit trasferiti è dato dalla dimensione del campo destinazione; la lunghezza del campo va specificata secondo la notazione seguente tutte le volte che non è altrimenti evidente
- Trasferimento di un dato di n bit: $\leftarrow_n$
  questa notazione si usa per trasferire un campo di n bit, tutte le volte che il numero di bit da trasferire non è evidente senza la relativa indicazione esplicita
- Contenuto di celle di memoria adiacenti a partire dall'indirizzo x: M[x]

Esempio:

$R1 \leftarrow_{32} M[x]$

indica il trasferimento verso R1 dei 4 byte M[x], M[[X+1], M[[X+2], M[[X+3]

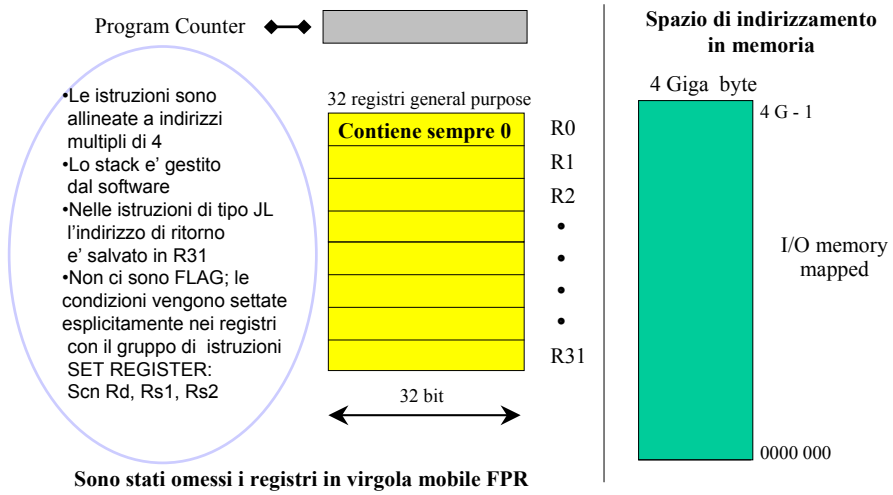
Calcolatori Elettronici L- A

L'ISA del processore DLX

Introduzione alla CPU “DLX”

- “DLX” è un nome di fantasia (sta per De LuXe). Questa CPU è stata progettata a scopo didattico da John Hennessy e da David Patterson (ne è stata realizzata una molto simile a questa a Stanford Univ., il “MIPS”)
- Contiene molte delle soluzioni progettuali adottate nelle CPU commerciali di oggi ma (essendo teorica) non obbliga all’approfondimento di complicati dettagli tecnici, inutili dal punto di vista didattico
- Load-Store cioè carica gli operandi dalla memoria ai registri ➔ esegue le operazioni ➔ salva nuovamente in memoria il risultato contenuto nel registro. E’ una macchina R-R le cui istruzioni assembler hanno formato (3-operandi : 0-indirizzi di memoria)
- Progettata con filosofia RISC: poche e semplici istruzioni nel LM per favorire l’implementazione della pipeline
- Spazio di I/O memory-mapped (assenza segnali di controllo per attivarlo)
- Assenza di istruzioni assembler per la gestione dello Stack
- contiene registri da 32bit di uso generale (intercambiabili nella sintassi del LM)
- Sia il Bus Dati che quello Indirizzi hanno 32 linee (4GB indirizzabili)
- 1 word = 32 bit

Ambiente di esecuzione nell'ISA del DLX R-R a 32 bit



Sono stati omessi i registri in virgola mobile FPR

Riassunto delle principali caratteristiche dell'ISA del DLX

- 32 registri da 32 bit di uso generale (GPR: R0..R31, con R0=0)
- 32 registri floating point da 32 bit (FPR: utilizzati esclusivamente dalle unità funzionali in virgola mobile)
- Istruzioni di lunghezza costante allineate a indirizzi multipli di 4
- 3 formati di istruzione: R, I, J
- Senza stack: non ci sono istruzioni specifiche, ma lo stack può essere realizzato via SW
- Nelle istruzioni di tipo JL (Jump & Link) l'indirizzo di ritorno e' salvato in R31
- Non c'è un registro di FLAG settato dalle istruzioni ALU; le condizioni vengono settate esplicitamente nei registri con il gruppo di istruzioni SET *condition* IN REGISTER: Scn Rd, Rs1, Rs2 (cn: EQ, LT, NEQ, …)
- Una sola modalità di indirizzamento (registro + offset)

Il set di istruzioni del DLX

- Le principali istruzioni aritmetiche e logiche
  - Istruzioni logiche anche con op. immediato: AND, ANDI, OR, ORI, XOR, XORI
  - Istruzioni aritmetiche: ADD, ADDI, SUB, SUBI, MULT, DIV
  - Istruzioni di traslazione logica (a destra anche aritmetica): SLL(I), SRL(I), SRA(I)
  - Istruzioni di SET CONDITION: Scn, ScnI, con cn = EQ, NE, LT, GT, LE, GE
- Le principali istruzioni di trasferimento dati
  - Load byte, Load Halfword, Load Word (LB, LH, LW)
  - Store byte, Store Halfword, Store Word (SB, SH, SW)
- Le principali istruzioni di trasferimento del controllo
  - Istruzioni di salto condizionato (PC+4 relative): BNEZ, BEQZ
  - Istruzioni di salto incondizionato diretto e indiretto(PC+4 relative): J, JR
  - Istruzioni di chiamata a procedura (Jump and Link, l'indirizzo di ritorno viene automaticamente salvato in R31): JAL, JALR
  - Istruzione di ritorno dalla procedura di servizio delle interruzioni: RFE

Istruzioni aritmetiche e logiche (istr. ALU)

- Istruzioni a 3 operandi
  - 2 operandi "*sorgente*"
  - 1 operando "*destinazione*".
- "*destinazione*": registro
- "*sorgente*": registro, registro / *operando immediato* (16 bit)

Esempi:

ADD R1, R2, R3 R1←R2+R3

ADDI R1, R2, 3 R1←R2+3

ADDI R1, R0, 5 R1←0+5 (cioè :R1← 5)

ADD R1, R5, R0 R1←R5+R0 (cioè : R1←R5)

Altre istruzioni di tipo ALU
SET CONDITION

Appartengono alla classe “ALU” anche le istruzioni di confronto.

Queste istruzioni confrontano i due operandi *sorgente* e mettono a “1” oppure a “0” l’operando *destinazione* in funzione del risultato del confronto

SEQ, =, set equal
SNE, ≠, set not equal
SLT, <, set less than
... (GE, LE, GE)

ESEMPI:

SLT R1,R2,R3 if (R2<R3): R1←1, else R1←0

Nel caso di operando sorgente immediato:

SLT**I** R1,R2,**5** if (R2<**5**): R1←1, else R1←0

Nel caso di confronto fra operandi “unsigned”:

SLT**U** R1,R2,R4 if (R2<R4): R1←1, else R1←0

Istruzioni di trasferimento dati LOAD/STORE

- Sono istruzioni che accedono alla memoria (load e store di byte, word e unsigned)
- L'indirizzo dell'operando in memoria è la somma del contenuto di un registro con un "offset" di 16 bit
- L'istruzione è codificata secondo il formato I (vedere più avanti)

Esempi:

LW R1, 40(R3) $R1 \leftarrow_{32} M[40+R3]$ ***"Sign Extension"***

LB R1, 40(R3) $R1 \leftarrow (M[40+R3]_7)^{24} \,\#\#\, M[40+R3]$

LBU R1,40(R3) $R1 \leftarrow (0)^{24} \,\#\#\, M[40+R3]$

SW 10(R5), R7 $M[10+R5] \leftarrow_{32} R7$

LHI R1, 42 $R1 \leftarrow 42 \,\#\#\, 0^{16}$

Istruzioni del trasferimento del controllo JUMP/JUMP & LINK

- "JUMP": Trasferimento del controllo (o salto) incondizionato
- "JUMP AND LINK: Trasferimento di controllo con salvataggio dell'indirizzo di ritorno (chiamata a procedura)
- "BRANCH": Trasferimento di controllo condizionato (salto condizionato, prossimo lucido)

"JUMP"

- J alfa PC ← (PC +4) + OFFSET(alfa) Formato Istruz. di tipo J...
  é "PC relative", l'offset è di 26 bit unsigned: $(PC+4)-2^{25}) \leq ofs(alfa) < (PC+4)+2^{25})$
- J R3 PC←R3 Formato Istruzione di tipo I...

"JUMP AND LINK"

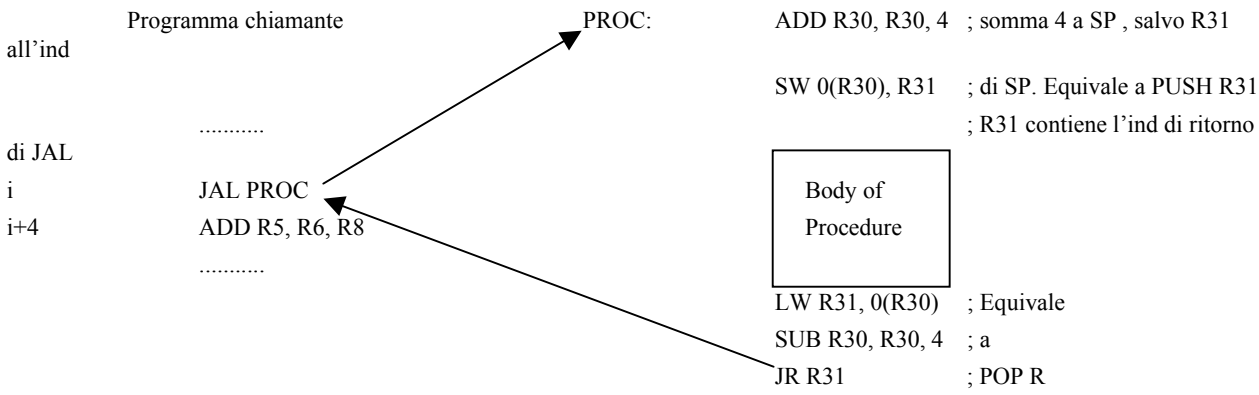
- JAL alfa R31←PC+4 poi PC ← (PC +4) + OFFSET(alfa); l'offset è di 26 bit Uns
- JALR R3 R31←PC+4 poi PC←R3
- JR31 PC←R31 è l'istruzione di ritorno da procedura (senza stack è possibile un solo livello di nesting)

Esempio: uno Stack per il nesting delle procedure nel DLX

- Un'area della memoria organizzata a Stack con un indirizzo di base e uno stack pointer è la struttura ideale per gestire le chiamate alle procedure. In taluni processori quest'area è addirittura definita nell'ISA. tanto che ad essa sono riservati registri speciali della CPU (ISA iA).
- Quando si chiama una procedura si salvano sullo stack l'indirizzo di quella che avrebbe dovuto essere la prossima istruzione da eseguire e i valori dei registri della CPU. Se all'interno della procedura ora in esecuzione c'è ulteriore necessità di trasferire (nesting) ancora il controllo prima di poter essere tornati al chiamante originale, si procede allo stesso modo con il salvataggio sullo stack. Mano a mano che si concludono le diramazioni vengono ripristinati i dati dallo stack ai registri (struttura LIFO) e si torna al chiamante originale mentre lo stack si svuota. Ma come si può implementare uno Stack se nell'ISA questo concetto non è previsto?

Lo faremo via software.

Useremo un registro, ad esempio R30, quale puntatore alla cima dello Stack incrementandolo o decrementandolo a seconda che dobbiamo aggiungere o prelevare dati dallo stack "virtuale". Jump And Link



Istruzioni del trasferimento del controllo BRANCH (salto condizionato)

"BRANCH"

- BEQZ = Branch Equal Zero
- BNEZ = Branch Not Equal Zero

- BEQZ R4, alfa if (R4=0): PC ← (PC +4)+OFFSET(alfa)

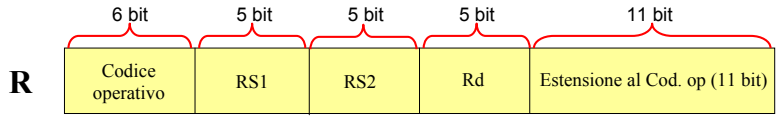
- BNEZ R4, alfa if (R4≠0): PC ← (PC +4)+OFFSET(alfa)

sono entrambe istruzioni "PC relative", con OFFSET di 16 bit con segno:
$(PC+4)-2^{15} \leq ofs(alfa) < (PC+4)+2^{15}$

BEQZ e BNEZ sono istruzioni di tipo I...

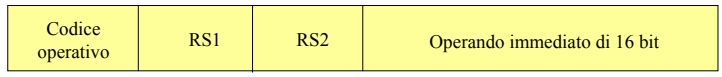
Con una istruzione di tipo SET CONDITION seguita da un'istruzione di salto condizionato si realizza la funzione di Compare and Branch (confronto e salto condizionato dal risultato del confronto) senza bisogno di flag dedicati

Formato tipico delle istruzioni nelle architetture R-R (Es. DLX)



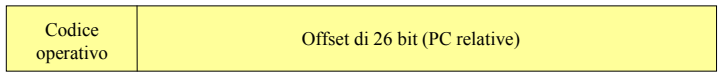
R

• Istruzioni aritmetiche e logiche del tipo Rd ← Rs1 op Rs2



I

• Load, Store, Branch condizionate, JR e JALR (trasf. controllo via registro), ALU con op. imm



J

• Trasferimento del controllo diretto e incondizionato (J e JAL)

Esempio di codice in Assembler di una macchina R-R (Es. DLX)

- Si scriva il codice assembler per il calcolo della somma degli elementi di un vettore A di 8 elementi e si utilizzino i seguenti registri:
  - **R1 per la somma corrente**
  - R2 per l'indice
  - R3 per il contatore delle iterazioni

```
        ADD     R1, R0, R0      ; azzera la somma corrente
        ADD     R2, R0, R0      ; R2 vale indice * 4  (4 e' numero byte per word)
        ADDI    R3, R0, 8       ; inizializza il contatore

CICLO   LW      R4, A(R2)       ; indirizzo dell'operando in memoria calcolato a run time
        ADD     R1, R1, R4      ; aggiorna somma corrente
        ADDI    R2, R2, 4
        ADDI    R3, R3, -1
        BNEZ    R3, CICLO
        SW      Z(R0), R1       ; Z e' la variabile con il risultato
        …..
```

Esercizi sul L.M. del DLX

- Con riferimento al lucido precedente si eseguano i seguenti esercizi:
  - si identifichi il formato di tutte le istruzioni
  - si disegni lo spazio di indirizzamento in memoria del DLX e si posizioni all'interno di esso il codice e i dati
  - Si disegni la dinamica del PC durante l'esecuzione del programma
  - Si elenchino i valori che R1 assume durante l'esecuzione del programma
  - Si calcoli il numero di cicli di bus che la CPU esegue durante l'esecuzione del programma suddividendoli in cicli di bus di FETCH e cicli di bus di EXECUTE

Calcolatori Elettronici L- A

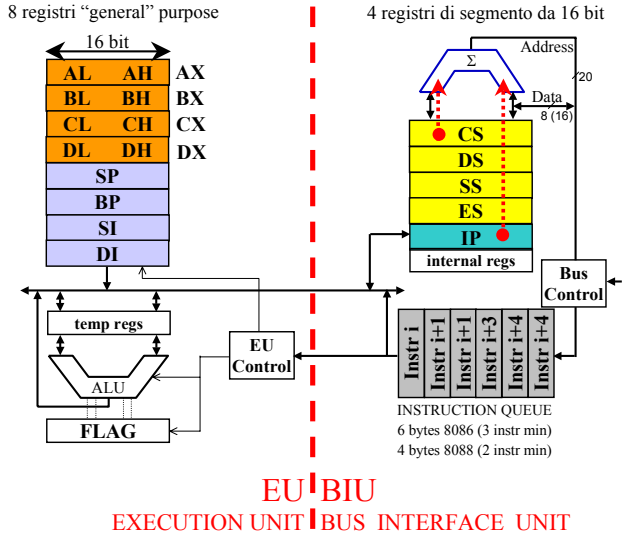
L'ISA dell'iA-16

ISA iA 16

Questa architettura prende il nome da “Architettura intel con bus dati a 16 bit”.

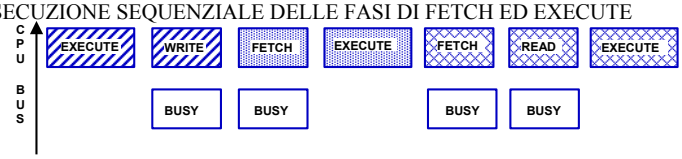
Concepita con l’obiettivo primario di aumentare il parallelismo consentendo il progetto di calcolatori a 16 bit (CPU 8086 con registri a 16 bit) è stata realizzata anche nella versione con bus dati a 8 bit (CPU 8088 con i medesimi registri e architettura interna praticamente identica) per realizzare calcolatori con bus dati a 8 bit in modo da facilitare l’interfacciamento con le periferiche a 8 bit, all’epoca dominanti il mercato.

ISA iA16
Architettura interna

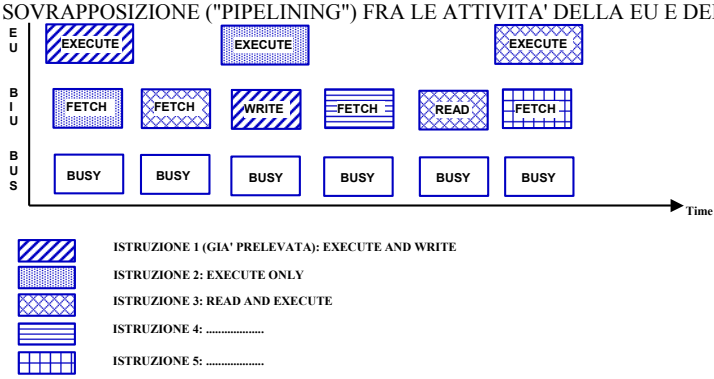


ISA iA16 - Sovrapposizione EU-BIU

CPU CON ESECUZIONE SEQUENZIALE DELLE FASI DI FETCH ED EXECUTE



CPU 8086/88: SOVRAPPOSIZIONE ("PIPELINING") FRA LE ATTIVITA' DELLA EU E DELLA BIU



Richiamo di aritmetica binaria

$1KB = 2^{10} = 1024_{10}$

$1MB = \text{mille } KB = 2^{10} \cdot 2^{10} = 2^{20}$ cioè deve essere "1" il bit con della ventesima potenza del 2, ossia il 21° bit (c'è anche 2^0 !):

$1\ 0000\ 0000\ 0000\ 0000\ 0000_2 = 1MB$

$1\ 0\ 0\ 0\ 0\ 0_{16}$

Per descrivere un indirizzo di memoria si inizia dalla locazione "0": per indirizzare 1MB di celle servono 1MB-1 indirizzi!
Infatti nell'8088 l'ultima locazione dello spazio fisico di 1MB ha indirizzo

$F\ F\ F\ F\ F_{16}$

che presuppone l'uso di 20 linee per uno spazio di 1MB.

Allo stesso modo, se intendiamo indirizzare uno spazio di memoria che parta dalla cella A (inclusa) e sia grande B locazioni, considereremo coinvolti gli indirizzi da A (inclusa) fino a A+B–1 (inclusa)

ISA iA16 - Organizzazione della memoria (1)

l'iA16 ha una architettura con memoria segmentata e segmenti da 64KB.

Per l'indirizzamento del MB sono necessarie 20 linee di indirizzo (indirizzo fisico). Siccome l'archit. interna è a 16 bit e c'era necessità di rispettare una compatibilità pregressa col DOS, le istruzioni ASM fanno uso di indirizzi con soli 16 bit che sarebbero sufficienti solo per indirizzare 64K elementi differenti...

Per potere quindi indirizzare tutto il MB si è introdotta la rappresentazione "logica" della memoria che fa uso di due parametri da **16 bit**: con la BASE (o SEGMENTO) si indirizza una delle 64K aree, mentre con l'OFFSET si identifica la cella all'interno dell'area selezionata che è contiene a sua volta 64K locazioni di memoria. Il programmatore "vede" attraverso l'ISA un insieme di 64K segmenti con dimensione massima 64KB.

Indirizzo Logico = (base, offset)

Quando il programmatore indica un indirizzo logico il processore lo converte in indirizzo fisico (detto anche Effective Address) per accedere alla memoria:

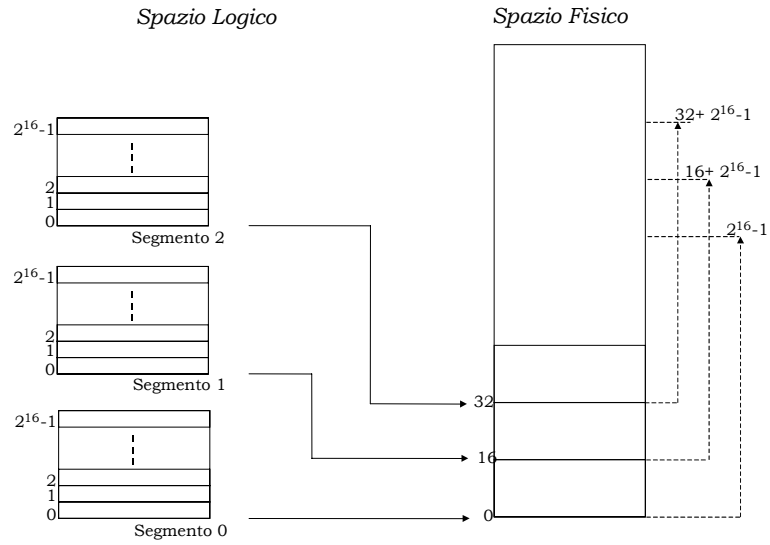
Indirizzo Fisico(20 bit) = base • 16 + offset

(il "• 16" si ottiene aggiungendo 4 zeri alla base espressa in binario)

Con questa tecnica è possibile indirizzare una memoria di dimensioni maggiori di 1MB e per questo motivo è possibile accedere alla stessa cella di memoria combinando diversamente i due parametri (sovrapposizione). Esempio (in base 10) di sovrapposizione:

Ind Logico		Ind Fisico
(1024,0)	→	16384
(128,14336)	→	16384

ISA iA16 - Organizzazione della memoria (2)

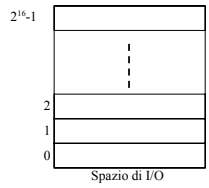


ISA iA16 - Organizzazione della memoria (3)

Lo **Spazio di I/O** è capace di 64K in tutto e non è segmentato:

Indirizzo Logico = Indirizzo Fisico

16 linee di indirizzo:



2^{16} (64K) locazioni di I/O

ISA iA16 - Registri interni

Registri “generali” (general purpose) purtroppo non “generici”:

AX, BX, CX, DX, BP, SP, SI, DI

Registri di segmento:

CS, DS, SS, ES

Registri di stato

IP, F

ISA iA16 - Registri Generali

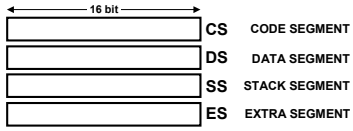
DATA REGISTERS POINTER & INDEX REGISTERS

• TUTTI I REGISTRI GENERALI POSSONO ESSERE USATI PER CONTENERE OPERANDI DELLE ISTRUZIONI
• INOLTRE CIASCUNO DI ESSI HA UN USO SPECIALE E VIENE USATO IMPLICITAMENTE DALLA CPU:

Registro	Operazione
AX	moltiplicazione, divisione e I/O a word (16bit)
AL	moltiplicazione, divisione e I/O a byte
AH	moltiplicazione, divisione byte
BX	generazione di offset
CX	operazioni su stringhe e loop
CL	rotazioni e shift di variabili
DX	moltiplicazione e divisione a word, I/O indiretto
BP	calcolo indirizzi
SP	operazioni sullo stack
SI	operazioni su stringhe
DI	operazioni su stringhe

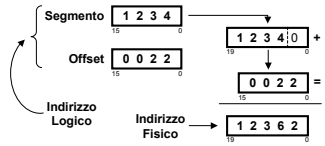
I registri del gruppo “pointer & index” vengono utilizzati per contenere valori che, opportunamente combinati, producono offset di operandi

ISA iA16 - Registri di Segmento (1)



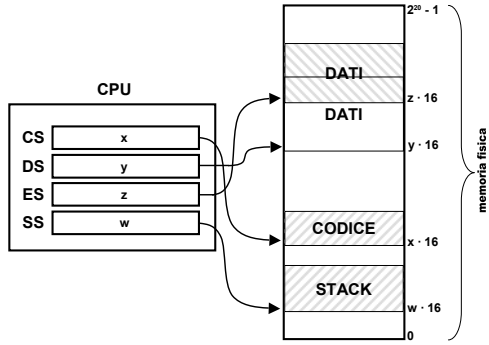
- I REGISTRI DI SEGMENTO VENGONO UTILIZZATI DALLA BIU PER IL CALCOLO DELL'INDIRIZZO FISICO

- OGNI REGISTRO DI SEGMENTO INDIRIZZA UN'AREA DI MEMORIA DI, AL MASSIMO, **64KB**
- I 4 SEGMENTI INDIVIDUATI DAL CONTENUTO DEI REGISTRI DI SEGMENTO VENGONO DETTI **SEGMENTI CORRENTI**
- **NELLE ISTRUZIONI VIENE INDICATO L'OFFSET**, MENTRE IL REGISTRO DI SEGMENTO UTILIZZATO E' AUTOMATICAMENTE DEFINITO IN BASE AL TIPO DI ISTRUZIONE
- IL PROGRAMMA, IN OGNI ISTANTE, **PUO' ACCEDERE SOLO A 4 SEGMENTI LOGICI** DISTINTI; PER ACCEDERE AD UN SEGMENTO LOGICO DIVERSO DAI SEGMENTI CORRENTI E' NECESSARIO MODIFICARE IL CONTENUTO DI UNO DEI 4 REGISTRI DI SEGMENTO



ISA iA16 - Registri di Segmento (2)

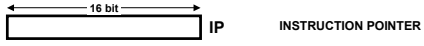
- ▶IL REGISTRO **CS** (CODE SEGMENT) PUNTA AL SEGMENTO IN CUI RISIEDONO LE ISTRUZIONI IN CORSO DI ESECUZIONE
- ▶I REGISTRI **DS** (DATA SEGMENT) E **ES** (EXTRA SEGMENT) PUNTANO AI SEGMENTI DATI CORRENTI
- ▶IL REGISTRO **SS** (STACK SEGMENT) PUNTA AL SEGMENTO CONTENENTE LO "STACK" CORRENTE



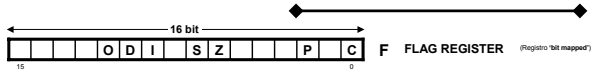
Ind Logico		Ind Fisico
(1024, 0)	➔	16384
(128, 14336)	➔	16384
(947, 1232)	➔	16384

- ▶L'INDIRIZZO FISICO CORRISPONDENTE ALL'INIZIO DI UN SEGMENTO (CIOE' ALL'INDIRIZZO DEL PRIMO BYTE DEL SEGMENTO STESSO) E' SEMPRE UN MULTIPLO DI 16
- ▶I SEGMENTI SI POSSONO "SOVRAPPORRE" IN MEMORIA
- ▶DI CONSEGUENZA, AD INDIRIZZI LOGICI DIVERSI PUO' CORRISPONDERE LO STESSO INDIRIZZO FISICO

ISA iA16 - Registri IP e Flag



- IL REGISTRO **IP** FORNISCE L'OFFSET DELLA PROSSIMA ISTRUZIONE DA ESEGUIRE NEL SEGMENTO DI CODICE CORRENTE
- LA COPPIA (CS, IP) E' IL PROGRAM COUNTER DELL'8088
- AL RESET VIENE POSTO CS=$0FFFF_H$, IP=0000_H
- LE ISTRUZIONI CHE PERMETTONO DI ALTERARE IL FLUSSO DI ESECUZIONE SEQUENZIALE (call, jmp, …) MODIFICANO IL CONTENUTO DEI REGISTRI **CS** E **IP**



	Flag	Condizione
Status flags e	**C** CARRY	1 se l'ultima istruzione ha generato un riporto "carry" o un prestito "borrow"
	P PARITY	1 se il risultato ha parità pari (viene testato solo il Byte meno significativo)
	Z ZERO	1 se il risultato dell'operazione è zero
	S SIGN	1 se il bit di segno del risultato è uguale a 1 (il risultato è un numero negativo)
	O OVERFLOW	1 se è avvenuto un traboccametno durante un calcolo aritmetico
Control	**I** INTERRUPT	se 1 abilita la CPU a riconoscere le richieste esterne di interruzione
	D DIRECTION	se 1 abilita il decremento nelle istruzioni che operano sulle stringhe

ISA linguaggi macchina 61

ZF-ZERO FLAG: vale 1 se il risultato vale 0.

SF-SIGN FLAG: vale 1 se il risultato è negativo.

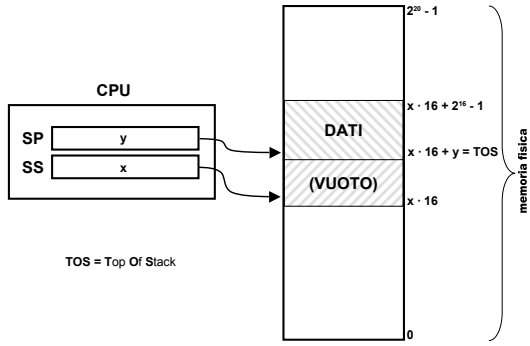
CF-CARRY FLAG: vale 1 se CO=1.

OF-OVERFLOW FLAG: vale 1 in caso di traboccamento del risultato di un'operazione fra numeri relativi rappresentati in complemento a 2

PF-PARITY FLAG: vale 1 se il bit di parità del risultato vale 1.

ISA iA16 - Lo Stack

- LO STACK (PILA) E' UN'AREA DI MEMORIA GESTITA DALLA CPU IN MODALITA' L.I.F.O. (LAST IN, FIRST OUT), E CIOE' IL DATO IMMESSO PER ULTIMO E' QUELLO PRELEVATO PER PRIMO
- IL REGISTRO **SS** PUNTA AL SEGMENTO UTILIZZATO COME STACK, E IL REGISTRO **SP** CONTIENE IN OGNI ISTANTE L'OFFSET DELLA LOCAZIONE RIEMPITA PER ULTIMA



TOS = Top Of Stack

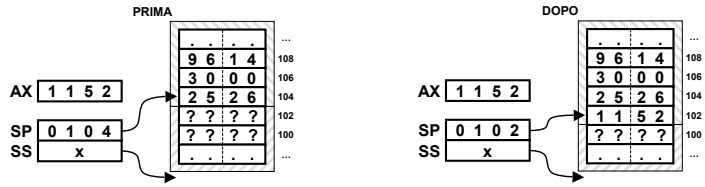
- LO STACK VIENE USATO DAL PROGRAMMA COME MEMORIA TEMPORANEA PER
  - •SALVATAGGIO DEL VALORE DI REGISTRI E VARIABILI
  - •PASSAGGIO DI PARAMETRI A PROCEDURE
  - •GESTIONE DI VARIABILI LOCALI DI PROCEDURE
- IN MEMORIA POSSONO ESSERE PRESENTI PIU' STACK, MA SOLO UNO E' ATTIVO IN OGNI MOMENTO, ED E' QUELLO ASSOCIATO AL VALORE CORRENTE DELLA COPPIA DI REGISTRI **(SS, SP)**
- LE ISTRUZIONI UTILIZZATE PER MEMORIZZARE E PRELEVARE DATI DALLO STACK SONO RISPETTIVAMENTE **PUSH** E **POP**

ISA iA16 - PUSH e POP sullo Stack

TUTTE LE OPERAZIONI DI IMMISSIONE E PRELIEVO HANNO OPERANDI A 16 BIT, CIOE' GESTISCONO SOLO **WORD**

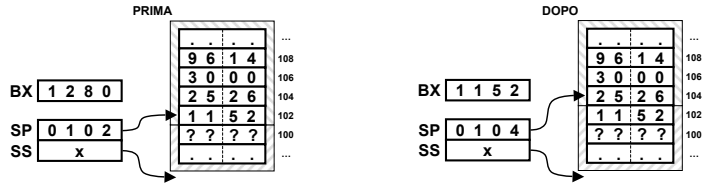
▶INSERIMENTO DI UN OPERANDO: Es. **PUSH AX**

SP VIENE DECREMENTATO DI 2, QUINDI L'OPERANDO VIENE INSERITO ALLA LOCAZIONE DI MEMORIA (SS, SP)



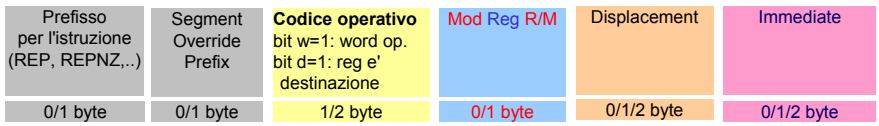
▶PRELIEVO DI UN OPERANDO: Es. **POP BX**

IL CONTENUTO DELLA LOCAZIONE DI MEMORIA (SS, SP) VIENE COPIATO NELL'OPERANDO, QUINDI IL REGISTRO **SP** VIENE INCREMENTATO DI 2



ISA iA16 - Formato Istruzioni (1)

Formato delle istruzioni dell'architettura iA16 (i8086/88 e dell'i80186/88) la cui lunghezza massima è 16 bit



Esempi di istruzioni assembler rappresentabili in codice binario con questo formato:

mov AX, alfa [BX + SI] ; **Cop Reg Disp Mod** R/**M**
add AX, BX ; **Cop Reg Mod R**/M
add BX, 48 ; **Cop Reg Op. Immediato**
add anno[BX+DI], 10 ; **Cop Mod** R/**M Disp Op. Immediato**

Questo stesso formato viene eseguito anche dalle architetture iA32 nei segmenti di codice con address-size da 16 bit (D=0)

ISA iA16 - Sintassi istruzioni (1)

[label] istruzione/direttiva [operando/i] [; commento]

label consente di dare un **nome simbolico** (da utilizzare come operando in altre istruzioni) a **variabili di memoria, valori, singole istruzioni, procedure**

istruzione/**direttiva** è il **mnemonico per un'istruzione o una direttiva**: individua il tipo di operazione da eseguire e il numero e il tipo degli operandi (la semantica dell'istruzione)

operando/i è una combinazione di nessuna, una o più **costanti**, **riferimenti a registri** o **riferimenti alla memoria**; se un'istruzione ammette due operandi:

- il primo è l'operando destinazione
- il secondo è l'operando sorgente

ad esempio, MOV AX, CX copia il contenuto del registro CX nel registro AX

commento consente di rendere più leggibile il programma

E' indifferente l'uso di lettere maiuscole o minuscole.

Il tipo di operandi ammessi varia da istruzione a istruzione.

Esistono istruzioni che ammettono come operando solo una costante o solo un particolare registro generale.

L'assembler dell'8086 non ha la caratteristica dell'ortogonalità, caratteristica che renderebbe la fase di apprendimento dell'assembler più veloce.

ISA iA16 - Sintassi istruzioni (2)

Nomi simbolici

Caratteri ammessi per i nomi simbolici:
A-Z a-z 0-9 _ $?

Il primo carattere di un nome non può essere un digit (0-9)

Ogni nome **deve essere univoco** (in genere, all'interno del modulo in cui viene definito)

Non deve coincidere con una parola riservata (ad esempio, il nome di un registro o di un operatore)

Costanti numeriche

Di default, tutti i valori numerici sono espressi **in base dieci**

è possibile esprimere le costanti numeriche:
-in base 16 (esadecimale) mediante il suffisso **H** (il primo digit deve però essere numerico)
-in base 8 (octal) mediante il suffisso **O**
-in base 2 (binario) mediante il suffisso **B** Ad esempio:

0FFh	
0h	
777O	
11001B	
FFh	errata !
778O	errata !

ISA iA16 - Direttive (1)

Istruzioni di tipo dichiarativo, che forniscono informazioni agli strumenti di sviluppo dei programmi (nel caso del Turbo Assembler, l'assemblatore TASM e il linker TLINK).

Direttive di segmento

Permettono di controllare in modo completo la definizione dei vari tipi di segmenti.

La definizione di ogni segmento deve iniziare con una **direttiva SEGMENT** e deve terminare con una **direttiva ENDS**:

```
nomeSeg SEGMENT
contenuto del segmento
nomeSeg ENDS
```

La direttiva SEGMENT definisce l'inizio di un segmento.

```
nome  SEGMENT
```

La direttiva ENDS definisce la fine di un segmento.

nome è il nome simbolico del segmento.

ISA iA16 - Direttive (2)

Direttive per la definizione di procedure

La definizione di ogni procedura deve iniziare con una direttiva PROC e terminare con una direttiva ENDP.
La direttiva PROC definisce l'inizio di una procedura.

nome PROC [distanza]

• **nome** è il **nome simbolico della procedura** (da utilizzare nelle chiamate alla procedura)
• **distanza** è:
-**NEAR** - la procedura può essere chiamata solo all'interno del segmento in cui è stata definita (default)
-**FAR** - la procedura può essere chiamata da qualsiasi segmento

La direttiva ENDP definisce la fine di una procedura.

nome ENDP

• **nome** è il nome simbolico della procedura.

Ad esempio, per definire la procedura FAR di nome FarProc, si scrive:

```
FarProc     PROC  FAR
            . . .
FarProc     ENDP
```

mentre per definire la procedura NEAR di nome NearProc, si scriverà:

```
NearProc    PROC  NEAR
            . . .
NearProc    ENDP
```

ISA iA16 - Direttive (3)

Direttive per la definizione dei dati

Permettono di definire:

il nome

il tipo

il contenuto delle variabili in memoria.

[nome] tipo espressione [, espressione] ...

nome - nome simbolico del dato

tipo - lunghezza del dato (se scalare) o di ogni elemento del dato (se array) - i tipi più usati sono:

DB	riserva uno o più byte (8 bit)
DW	riserva una o più word (16 bit)
DD	riserva una o più doubleword (32 bit)

espressione - contenuto iniziale del dato:

- un' espressione costante
- una stringa di caratteri (solo DB)
- un punto di domanda (nessuna inizializzazione)
- un'espressione che fornisce un indirizzo (solo DW e DD)
- un'espressione **DUP**licata

ISA iA16 - Direttive (4)

Esempi di uso di Direttive per la definizione dei dati

```
ByteVar    DB      0             ; 1 byte inizializzato a 0
ByteArray  DB      1,2,3,4       ; array di 4 byte
String     DB      '8','0','8','6'  ; array di 4 caratteri
String     DB      '8086'        ; equivale al precedente
Titolo     DB      'Titolo',0dh,0ah;  stringa che contiene anche una
                                      ; coppia di caratteri CR/LF
Zeros      DB      256 dup (0);  array di 256 byte inizializzati a 0
Tabella    DB      50 dup (?)  ; array di 50 byte non inizializzati
WordVar    DW      100*50      ; scalare di una word
Matrix     DW      1,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0 ;  array di 16 word
Matrix     DW      4 dup (1, 3 dup (0)) ;  equivale al precedente
NearPointer        DW      Matrix ;  contiene l'offset di Matrix
DoubleVarDD        ?       ; scalare di una doubleword
FarPointer DD      Matrix  ; contiene l'offset e l'indirizzo del
                           ; segmento di Matrix
```

PER OGNI VARIABILE SI SPECIFICANO NOME, TIPO E VALORE INIZIALE

ISA iA16 - Tipi di istruzioni

- ISTRUZIONI OPERATIVE
  - ISTRUZIONI PER IL TRASFERIMENTO DI DATI
  - ISTRUZIONI LOGICO-ARITMETICHE
  - ISTRUZIONI DI TRASLAZIONE E ROTAZIONE
  - ISTRUZIONI PER LA MODIFICA DEI FLAG
- ISTRUZIONI PER IL CONTROLLO DEL FLUSSO DI ESECUZIONE
  - ISTRUZIONI DI SALTO (CONDIZIONATO ED INCONDIZIONATO)
  - ISTRUZIONI PER LA GESTIONE DELLE PROCEDURE
  - ISTRUZIONI PER L'ATTIVAZIONE DEGLI INTERRUPT SOFTWARE

Il Set di Istruzioni dell'iA 16

- Le principali istruzioni aritmetiche e logiche
  - Istruzioni logiche anche con op. immediato: AND, OR, XOR, NOT
  - Istruzioni aritmetiche: ADD, ADC, SUB, SUBB, INC, DEC, NEG, CMP, MUL, IMUL, DIV,IDIV
  - Istruzioni di traslazione logica e aritmetica: SAL, SAR, SHL, SHR
- Le principali istruzioni di trasferimento dati
  - MOV tra due registri Es.: mov ax, bx ax ← bx
  - MOV tra registro e memoria Es: mov al, alfa[bx+si] al ← M[(DS<<4) + bx + si + alfa]
  - PUSH, POP, PUSHF, POPF Es.: push ax M[SS:SP] ← AX; SP ← SP - 2
- Le principali istruzioni di trasferimento del controllo
  - Istruzioni di salto condizionato (PC+n relative): Jxx, es. JC, JNC, JE, JNE, JZ, JNZ, BEQZ
  - Istruzioni di salto incondizionato: JMP (nello stesso segmento o NEAR e tra segmenti diversi o FAR)
  - Istruzione di chiamata a procedura: CALL, l'indirizzo di ritorno viene automaticamente salvato in stack
  - istruzione di ritorno da procedura: RET
  - Istruzione di ritorno dalla procedura di servizio delle interruzioni: IRET

ISA iA16 - Modalità di indirizzamento

- IL NOME DI UNA VARIABILE RAPPRESENTA L'OFFSET DELLA VARIABILE NEL SEGMENTO IN CUI E' DEFINITA x = 0, y = 1, z= 3

- INDIRIZZAMENTO DIRETTO
  SE L'IDENTIFICATORE NUMERICO (SEGMENTO LOGICO) CHE IL LINKER ASSOCIA A dati E' STATO CARICATO IN UNO DEI REGISTRI DI SEGMENTO ALLORA E' POSSIBILE ACCEDERE ALLE VARIABILI DI dati TRAMITE IL LORO NOME

ad esempio se DS ← SEG dati :
MOV AH,x , ADD BX,y , MOV z,CX , MOV z+2,BP

L'ARCHITETTURA iA16 SUPPORTA MODALITA' DI INDIRIZZAMENTO PIU' COMPLESSE. NELL'ESAME DEL SET DI ISTRUZIONI FAREMO RIFERIMENTO AL SOLO INDIRIZZAMENTO DIRETTO.

ISA iA16 - Istruzione MOV (1)

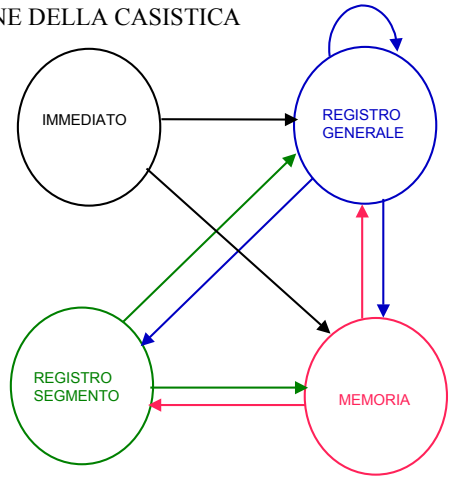
MOV destination,source

- AZIONE:
  SOSTITUISCE L'OPERANDO DESTINAZIONE CON UNA COPIA DELL'OPERANDO SORGENTE
- CASISTICA ED ESEMPI

memoria,registro generale:	MOV x,AL
registro generale, memoria:	MOV BX,y
registro generale, registro generale:	MOV AX, DX
memoria,immediato:	MOV y,FFFFH
registro generale,immediato:	MOV BP, A3FFH
memoria,registro segmento:	MOV y, DS
registro segmento, memoria :	MOV ES, y
registro segmento, registro generale:	MOV DS, CX
registro generale, registro segmento:	MOV SI, ES

ISA iA16 - Istruzione MOV (2)

SCHEMATIZZAZIONE DELLA CASISTICA



NON SONO QUINDI POSSIBILI I TRASFERIMENTI
MEM⇔MEM, IMM⇔SEGM REG, SEGM REG⇔SEGM REG

ISA iA16 - Istruzione PUSH

PUSH source

- AZIONE:
  SALVA SULLO STACK UNA COPIA DELL'OPERANDO SORGENTE
  (L'OPERANDO DEVE ESSERE A 16 BIT). PIU' IN DETTAGLIO:
  1) IL CONTENUTO DI SP VIENE DECREMENTATO DI DUE
  2) LA COPIA DELL'OPERANDO SORGENTE VIENE MEMORIZZATA
  ALL'INDIRIZZO (SS,SP)

  - CASISTICA ED ESEMPI:

memoria:	PUSH y
immediato:	PUSH 345AH
registro generale:	PUSH AX
registro di segmento:	PUSH DS

ISA iA16 - Istruzione POP

POP destination

- AZIONE:
  RIMUOVE DALLA PILA UNA WORD E LA COPIA NELL'OPERANDO DESTINAZIONE (L'OPERANDO DEVE ESSERE A 16 BIT)
  PIU' IN DETTAGLIO:
  1) L'OPERANDO DESTINAZIONE VIENE SOSTITUITO DALLA WORD MEMORIZZATA ALL'INDIRIZZO (SS,SP)
  2) IL CONTENUTO DI SP VIENE INCREMENTATO DI DUE, RIMUOVENDO COSI' DALLO STACK LA WORD COPIATA NELL'OPERANDO DESTINAZIONE
  - CASISTICA ED ESEMPI

    memoria: POP y

    registro generale: POP AX

    registro di segmento: POP DS

ISA iA16 - Istruzioni di I/O

IN accumulatore,porta OUT porta,accumulatore

- AZIONE:
  ESEGUONO I TRASFERIMENTI DATI FRA PROCESSORE E SPAZIO DI I/O

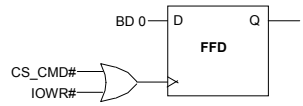
- CASISTICA ED ESEMPI
  SE LA PORTA HA INDIRIZZO <= 255 SI PUO' USARE L'INDIRIZZAMENTO DIRETTO:
  IN AL, 1AH IN AX, 80H
  OUT FFH, AL OUT FFH, AX

  SE LA PORTA HA INDIRIZZO > 255 SI DEVE USARE L'INDIRIZZAMENTO TRAMITE IL REGISTRO DX:
  IN AL, DX IN AX, DX
  OUT DX, AL OUT DX, AX

ISA iA16 - Esempio Istruzioni di I/O

Esempio:

Si scriva il codice per comandare (accendere e spegnere) una periferica utilizzando la porta di I/O di indirizzo 0FFH.




```
COMMAND     EQU     0FFH            ; assegna FF a COMMAND
            MOV  AL, 01H            ;
            OUT  COMMAND, AL        ; accende la periferica
            MOV  AL, 0H             ;
            OUT  COMMAND, AL        ; spegne la periferica
```

Il flip-flop risulta mappato all'indirizzo di I/O FFH. Scrivendo un "1" a questo indirizzo per mezzo della OUT si attivano sia la decodifica dell'indirizzo CS_CMD# che il segnale IOWR# la cui contemporaneità causa il campionamento dell'1 da parte del FF che accende la periferica collegata alla sua uscita Q. Per lo spegnimento la procedura è analoga e basta caricare uno zero in AL prima di eseguire la OUT. Il segnale di comando per la periferica sarà modificato solo dagli accessi alla porta a lei riservata di indirizzo FFH.)

ISA iA16 - Istruzioni di Addizione

ADD destination,source

- AZIONE:

MODIFICA L'OPERANDO DESTINATARIO SOMMANDOVI L'OPERANDO SORGENTE. METTE A 1 IL FLAG CF SE, INTERPRETANDO GLI OPERANDI COME NUMERI NATURALI, SI VERIFICA UN RIPORTO. METTE A 1 FLAG OF SE, INTERPRETANDO GLI OPERANDI COME NUMERI INTERI, SI VERIFICA UN TRABOCCAMENTO.

ADC destination,source

- AZIONE:

MODIFICA L'OPERANDO DESTINATARIO SOMMANDOVI SIA L'OPERANDO SORGENTE CHE IL FLAG CF. LE AZIONI SUI FLAG CF E OF E LA CASISTICA SONO IDENTICHE ALL'ISTRUZIONE ADD.

- CASISTICA ED ESEMPI:

memoria,registro gen:	ADD x,AL
registro gen, memoria:	ADD AX,y
registro gen, registro gen:	ADD AX, DX
memoria, immediato:	ADD y, FFFFH
registro gen, immediato:	ADD BP, A3FFH

ISA iA16 - Incremento Decremento

INC destination

- AZIONE:
  MODIFICA L'OPERANDO
  DESTINATARIO
  SOMMANDOVI 1
- CASISTICA ED ESEMPI

  memoria: INC y

  registro generale: INC AL

DEC destination

- AZIONE:
  MODIFICA L'OPERANDO
  DESTINATARIO SOTTRAENDOVI 1
- CASISTICA ED ESEMPI

  memoria: DEC y

  registro generale: DEC AL

ISA iA16 - Istruzioni di Sottrazione

SUB destination,source

- AZIONE:
  MODIFICA L'OPERANDO DESTINATARIO SOTTRAENDOVI L'OPERANDO SORGENTE. METTE A 1 IL FLAG CF SE, INTERPRETANDO GLI OPERANDI COME NUMERI NATURALI, SI VERIFICA UN PRESTITO. METTE A 1 FLAG OF SE, INTERPRETANDO GLI OPERANDI COME NUMERI INTERI, SI VERIFICA UN TRABOCCAMENTO.

SBB destination,source

- AZIONE:
  MODIFICA L'OPERANDO DESTINATARIO SOTTRAENDOVI SIA L'OPERANDO SORGENTE CHE IL FLAG CF. LE AZIONI SUI FLAG CF E OF E LA CASISTICA SONO IDENTICHE ALL'ISTRUZIONE ADD.

- CASISTICA ED ESEMPI

memoria,registro generale:	SUB x,AL
registro generale, memoria:	SUB AX,y
registro gen, registro gen:	SUB AX, DX
memoria,immediato:	SUB y, FFFFH
registro generale,immediato:	SUB BP, A3FFH

ISA iA16 - Istruzioni di Moltiplicazione

MUL source

- AZIONE:
  MODIFICA L'OPERANDO DESTINATARIO (IMPLICITO) SOSTITUENDO AD ESSO IL PRODOTTO DI OPERANDO DESTINATARIO E OPERANDO SORGENTE. GLI OPERANDI SONO INTERPRETATI COME NUMERI NATURALI.
  PIU' IN DETTAGLIO:
  SE L'OPERANDO SORGENTE E' A 8 BIT
  AX ← AL*SOURCE
  SE L'OPERANDO SORGENTE E' A 16 BIT
  DX, AX ← AX*SOURCE

IMUL source

- AZIONE:
  L'UNICA DIFFERENZA CON L'ISTRUZIONE MUL E' CHE GLI OPERANDI SONO INTERPRETATI COME NUMERI INTERI

- CASISTICA ED ESEMPI:

  memoria: MUL x

  registro generale: MUL CX

ISA iA16 - Divisione

DIV source

- AZIONE:
  CONSIDERA L'OPERANDO SORGENTE COME DIVISORE E L'OPERANDO DESTINATARIO (IMPLICITO) COME DIVIDENDO ED EFFETTUA L'OPERAZIONE DI DIVISIONE. GLI OPERANDI SONO INTERPRETATI COME NUMERI NATURALI.
  PIU' IN DETTAGLIO:
  SE L'OPERANDO SORGENTE E' A 8 BIT
  AL ← QUOZIENTE DI AX / SOURCE
  AH ← RESTO DI AX / SOURCE
  SE L'OPERANDO SORGENTE E' A 16 BIT
  AX ← QUOZIENTE DI (DX,AX) / SOURCE
  DX ← RESTO DI (DX,AX) / SOURCE

IDIV source

- AZIONE:
  L'UNICA DIFFERENZA CON L'ISTRUZIONE DIV E' CHE GLI OPERANDI SONO INTERPRETATI COME NUMERI INTERI

- CASISTICA ED ESEMPI:
  memoria: DIV y

  registro generale: DIV CL

ISA iA16 - Confronto di due operandi

CMP destination,source

- AZIONE:
  VERIFICA SE L'OPERANDO DESTINATARIO E' MAGGIORE, UGUALE O MINORE DELL' OPERANDO SORGENTE. GLI OPERANDI SONO INTERPRETATI SIA COME NUMERI NATURALI CHE COME NUMERI INTERI. L'ISTRUZIONE AGGIORNA IL CONTENUTO DEL REGISTRO DEI FLAG IN BASE AL RISULTATO DELLA VERIFICA E LASCIA INALTERATI ENTRAMBI GLI OPERANDI. L'AGGIORNAMENTO E' CONSISTENTE CON L'INTERPRETAZIONE DEL CONTENUTO DEL REGISTRO DEI FLAG CHE SARA' DATA DALL'ISTRUZIONE DI SALTO CONDIZIONATO CHE IN UN PROGRAMMA NORMALMENTE SEGUE SEMPRE L'ISTRUZIONE CMP.

  - CASISTICA ED ESEMPI:

memoria,registro generale:	CMP x,AL
registro generale, memoria:	CMP AX,y
registro generale, registro generale:	CMP SI, DI
memoria,immediato:	CMP y, FFFFH
registro generale,immediato:	CMP DX, 0

ISA iA16 - Istruzioni di tipo Logico (ALU)

logic_op destination,source

- AZIONE:
  SOSTITUISCE CIASCUN BIT DELL' OPERANDO DESTINATARIO CON IL RISULTATO DELL'OPERAZIONE LOGICA *logic_op* TRA IL BIT STESSO ED IL BIT CORRISPONDENTE DELL'OPERANDO SORGENTE.
  logic_op : AND, OR, XOR.

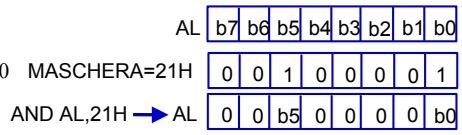
  - CASISTICA ED ESEMPI

memoria,registro generale:	AND x,AL
registro generale, memoria:	OR AX,y
registro generale, registro generale:	XOR AX, DX
memoria,immediato:	AND y, 00FFH
registro generale,immediato:	XOR CX, FFFFH

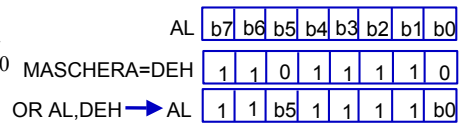
ISA iA16 - Operazioni su singoli bit (Bit Masking)

L'USO DI ISTRUZIONI LOGICHE CON OPPORTUNE MASCHERE CONSENTE DI MODIFICARE UNO O PIU' BIT DELL'OPERANDO DESTINATARIO

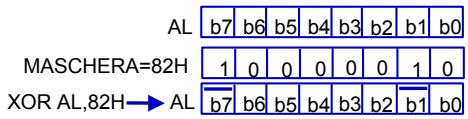
- FORZARE A 0 TUTTI I BIT DI AL LASCIANDO INALTERATI I BIT 0 E 5 : **AND AL,21H**



- FORZARE A 1 TUTTI I BIT DI AL LASCIANDO INALTERATI I BIT 0 E 5 : **OR AL,DEH**



- COMPLEMENTARE I BIT 1 E 7 DI AL LASCIANDO INALTERATI GLI ALTRI BIT: XOR **AL,82H**

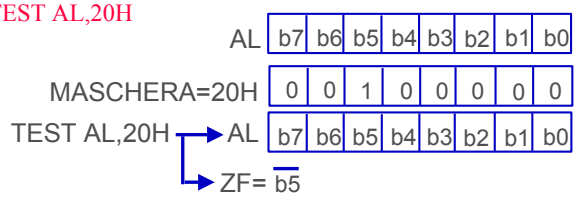


ISA iA16 - Istruzione TEST

TEST destination,source

- AZIONE:
  ESEGUE L'AND DEI DUE OPERANDI AGGIORNANDO I FLAG
  IN FUNZIONE DEL RISULTATO. NON MODIFICA L'OPERANDO DESTINAZIONE
  (e' l'equivalente "logico" della CMP). LA CASISTICA E' QUELLA DELLE
  ISTRUZIONI DI TIPO LOGICO.

- UN ESEMPIO:
  VERIFICARE IL VALORE ASSUNTO DAL BIT 5 DI AL SENZA MODIFICARE
  AL: TEST AL,20H

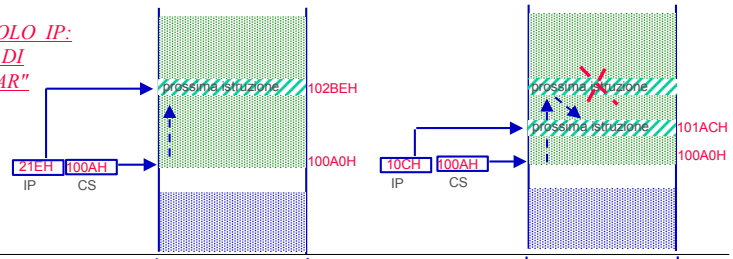


L'ISTRUZIONE TEST SARA' SEGUITA DA UN' ISTRUZIONE DI SALTO
CONDIZIONATO CHE FA EVOLVERE IL PROGRAMMA IN FUNZIONE DEL
VALORE DI ZF (JZ,JNZ)

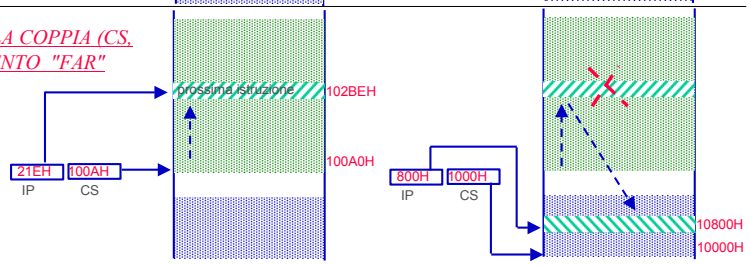
ISA iA16 - Controllo di flusso

- (CS,IP) : PROGRAM COUNTER DELL'ARCHITETTURA iA16
- CS PUNTA AL SEGMENTO DI CODICE CORRENTE, IP CONTIENE L'OFFSET DELLA PROSSIMA ISTRUZIONE DA ESEGUIRE
- LE ISTRUZIONI DI CONTROLLO DI FLUSSO MODIFICANO IL VALORE DI IP O DELLA COPPIA (CS,IP)

- *MODIFICA DEL SOLO IP: TRASFERIMENTO DI CONTROLLO "NEAR"*



- *MODIFICA DELLA COPPIA (CS, IP): TRASFERIMENTO "FAR"*



ISA linguaggi macchina 89

ISA iA16 - Salto condizionato (1)

- LE ISTRUZIONI DI SALTO CONDIZIONATO (JE,JNE,JA...) TRASFERISCONO O NON TRASFERISCONO IL CONTROLLO IN DIPENDENZA DELLO STATO DEL REGISTRO DEI FLAG. I SALTI CONDIZIONATI PROVOCANO DEI TRASFERIMENTI DI CONTROLLO NELL'AMBITO DEL SEGMENTO DI CODICE CORRENTE: SE LA CONDIZIONE DI SALTO E' VERIFICATA VIENE MODIFICATO IL SOLO IP.
- LE ISTRUZIONI DI SALTO CONDIZIONATO SONO LUNGHE 2 BYTE; IL PRIMO BYTE E' L'OPCODE ASSOCIATO AL TIPO DI SALTO, IL SECONDO BYTE E' UN INTERO IN COMPLEMENTO A 2 CHE RAPPRESENTA LO SPIAZZAMENTO DEL TARGET DEL SALTO RISPETTO AL VALORE CORRENTE DI IP (IL TARGET DEL SALTO E' SPECIFICATO DAL PROGRAMMATORE MEDIANTE UNA LABEL CHE SEGUE IL MNEMONICO DEL SALTO).
- SE LA CONDIZIONE DI SALTO E' VERIFICATA LO SPIAZZAMENTO VIENE SOMMATO AD IP. E' POSSIBILE QUINDI RAGGIUNGERE SOLO TARGET APPARTENENTI AL SEGMENTO CORRENTE E DI OFFSET COMPRESO FRA IP-128 E IP+127
- I SALTI CONDIZIONATI VENGONO DETTI "SHORT" PERCHE' PERMETTONO DI RAGGIUNGERE SOLO TARGET "VICINI" E "PC-RELATIVE" PERCHE' SI BASANO SULLO SPIAZZAMENTO DEl TARGET RISPETTO AL PROGRAM COUNTER INVECE CHE SUL SUO INDIRIZZO ASSOLUTO. (PC-RELATIVE ⟶ ROUTINES "POSITION INDEPENDENT")

ISA iA16 - Salto condizionato (2)

J*cond* label

- AZIONE:
  ESAMINA IL REGISTRO DEI FLAG PER VERIFICARE SE E' SODDISFATTA LA CONDIZIONE SPECIFICATA MEDIANTE *cond.* SE LA CONDIZIONE E' VERIFICATA IL CONTROLLO VIENE TRASFERITO ALL'ISTRUZIONE ASSOCIATA A *label*, SE NON E' VERIFICATA LA CPU ESEGUE L'ISTRUZIONE SUCCESSIVA. UN SALTO CONDIZIONATO SEGUE TIPICAMENTE UNA CMP CHE CONFRONTA DUE OPERANDI E AGGIORNA I FLAG.

CASISTICA ED ESEMPI:

dopo aver eseguito ad esempio CMP AX, 3 si esegue:

JE alfa;salta alla label "alfa" se AX=3 (JE: jump if equal)

JNE alfa;salta alla label "alfa" se AX diverso da 3 (JNE: jump if not equal)

JA alfa;salta alla label "alfa" se AX>3 (JA: jump if above)

JB alfa;salta alla label "alfa" se AX<3 JB: jump if below)

JAE alfa;salta alla label "alfa" se AX>=3 (JE: jump if above or equal)

JBE alfa;salta alla label "alfa" se AX <= 3 (JBE: jump if below or equal)

Esistono istruzioni corrispondenti a JA, JB, JAE, JBE per il confronto tra interi con segno:

JG (jump if greater)

JL (jump if less)

JGE (jump if greater or equal)

JLE (jump if less or equal)

continua...

ISA iA16 - Salto condizionato (3)

Altre istruzioni di salto:
JZ alfa ;salta alla label "alfa" se il risultato dell'istruz precedente e' 0 (JZ: jump if zero)
JNZ alfa ;salta alla label "alfa" se il risultato dell'istruz preced e' diverso da 0 (JNZ: jump if not zero)
JC/JNC beta ;salta alla label "beta" se il l'ultima istruzione ha messo a 1/0 il CF (jump if carry / not carry)
JO/JNO beta ;salta alla label "beta" se il l'ultima istruzione ha messo a 1/0 il flag OF (jump if overflow / not overflow)

- UN ESEMPIO PIU' COMPLETO: LETTURA DI DUE DATI DA DUE BATTERIE DI SWITCH E ACCENSIONE DI UNA BATTERIA DI LED NEL CASO DI DATI UGUALI

```
            IN AL,10H     ;lettura primo dato all'indirizzo 10H
            MOV BL,AL     ;salvataggio in BL del dato precedente
            IN AL,20H     ;lettura secondo dato all'indirizzo 20H
            CMP AL,BL     ;confronto fra i due dati
            JE uguali     ;salto alla label "uguali" in caso di dati uguali
            MOV AL,0      ;il dato 0 spegne i led
            JMP invio     ;salto assoluto (non-condizionato) all'istruzione di out
uguali:     MOV AL,FFH    ;il dato FFH accende i led
invio:      OUT 15H,AL    ;accensione/spegnimento dei led all'indirizzo 15H
```

ISA iA16 - Salti assoluti (1)

JMP target

JMP TRASFERISCE IL CONTROLLO ALL'ISTRUZIONE TARGET.

- JMP DIRETTO: IL TARGET DEL SALTO E' SPECIFICATO IN MODO ESPLICITO
- JMP INDIRETTO: SE IL TARGET DEL SALTO E' SPECIFICATO MEDIANTE UN PUNTATORE (REGISTRO-VARIABILE IN MEMORIA)

JMP DIRETTO: JMP label

label "near": JMP di tipo near

2 o 3 byte: opcode + disp. (1 o 2 byte)

formato a 2 byte: jmp short sempre PC-relative

```
code1 SEGMENT
.
JMP beta
.
beta: MOV AX,CX
.
code1 ENDS
```

IP=IP+displacement

label "far": JMP di tipo far

5 byte: opcode + OFFSET label+SEG label

```
code1 SEGMENT
.
JMP beta
.
code1 ENDS

code2 SEGMENT
.
 beta LABEL FAR
 MOV AX,CX
.
code2 ENDS
```

CS=code2 , IP=OFFSET(beta)

ISA iA16 - Salti assoluti (2)

JMP INDIRETTO: JMP pointer

JMP di tipo near

a) pointer=registro generale

```
.
JMP AX
.
```

IP=AX

b) pointer=variabile (tipo word)

```
 dati SEGMENT
.
y WORD OFFSET beta
.
dati ENDS

code1 SEGMENT
.
JMP y
.
beta: MOV AX,CX
.
code1 ENDS
```

IP= valore di y

JMP di tipo far

pointer=variabile (tipo dword)

```
dati SEGMENT
.
z DWORD beta
.
dati ENDS

code1 SEGMENT
.
JMP z
.
code1 ENDS
.
code2 SEGMENT
.
beta: MOV AX,CX
.
code2 ENDS
```

(CS,IP) = valore di z :
CS=word meno significativa
IP= word piu' significativa

ISA iA16 - Istruzione LOOP

LOOP label

DECREMENTA CX SENZA MODIFICARE I FLAG E SE DOPO IL DECREMENTO CX E' DIVERSO DA 0 SALTA ALLA LABEL. LOOP PERMETTE COSI' DI REALIZZARE DEI CICLI DI TIPO "for" IN CUI CX HA LA FUNZIONE DI CONTATORE DELLE ITERAZIONI.
UN ESEMPIO:
Ripetizione per 10 volte di un blocco di codice che legge due valori dalle porte di input 388H e 39CH e invia sulla porta di output 380H il maggiore dei due valori.

IL TRASFERIMENTO DI CONTROLLO E' SEMPRE DI TIPO SHORT (formato a 2 byte con opcode+disp.)

```
        MOV CX,10
ripeti: MOV  DX,388H
        IN AL,DX
        MOV BL,AL
        MOV  DX,39CH
        IN AL,DX
        CMP AL,BL
        JAE maggiore  ;salta se AL ≥ BL
        MOV AL,BL     ;eseguita solo se AL<BL
maggiore: MOV  DX,380H
        OUT DX,AL
        LOOP ripeti
```

ISA iA16 - Chiamata a procedura CALL (1)

CALL procedura

LA CHIAMATA A PROCEDURA E' UN TRASFERIMENTO DI CONTROLLO ASSOLUTO CHE, A DIFFERENZA DI JMP, SALVA IN STACK L'INDIRIZZO DELL'ISTRUZIONE IMMEDIATAMENTE SUCCESSIVA, PERMETTENDO DI RITRASFERIRE IL CONTROLLO A QUEST'ULTIMA DOPO L'ESECUZIONE DELLA PROCEDURA. LA CASISTICA E' IDENTICA A QUELLA DELL'ISTRUZIONE JMP (l'unica differenza e' che non esistono CALL di tipo short).

CALL DIRETTA: CALL label

label "near": CALL di tipo near

3 byte: opcode+disp-l+disp-h sempre PCrelat

```
code1 SEGMENT
.
proc1 PROC NEAR
.
proc1 ENDP
.
CALL proc1
.
code1 ENDS
```

PUSH IP
IP=IP+displacement

label "far": CALL di tipo far

5 byte: opcode + OFFSET label+ SEG label

```
code1 SEGMENT
.
proc1 PROC FAR
.
proc1 ENDP
.
code1 ENDS

code2 SEGMENT
.
CALL proc1
.
code2 ENDS
```

PUSH CS, PUSH IP
CS=code1 , IP=OFFSET proc1

ISA iA16 - Chiamata a procedura CALL (2)

CALL INDIRETTA: CALL pointer

CALL di tipo near

a) pointer=registro generale

CALL AX

PUSH IP
IP=AX

oppure

b) pointer=variabile (tipo word)

```
dati SEGMENT
.
y WORD OFFSET proc1
.
dati ENDS
.
code1 SEGMENT
.
CALL y
.
proc1 PROC NEAR
.
proc1 ENDP
.
code1 ENDS
```

PUSH IP
IP= valore di y

CALL di tipo far

pointer=variabile (tipo dword)

```
dati SEGMENT
.
z DWORD proc1
.
dati ENDS

code1 SEGMENT
.
CALL z
.
code1 ENDS
.
code2 SEGMENT
.
proc1 PROC FAR
.
proc1 ENDP
.
code2 ENDS
```

PUSH CS, PUSH IP
CS=word meno significativa di z
IP= word piu' significativa di z

ISA iA16 - Ritorno da procedura

RET

L'ISTRUZIONE RET EFFETTUA IL RITORNO DA UNA PROCEDURA AL PROGRAMMA CHIAMANTE AGGIORNANDO IP (O LA COPPIA CS,IP) CON LA WORD (O LA DOUBLE WORD) PRELEVATA DALLO STACK

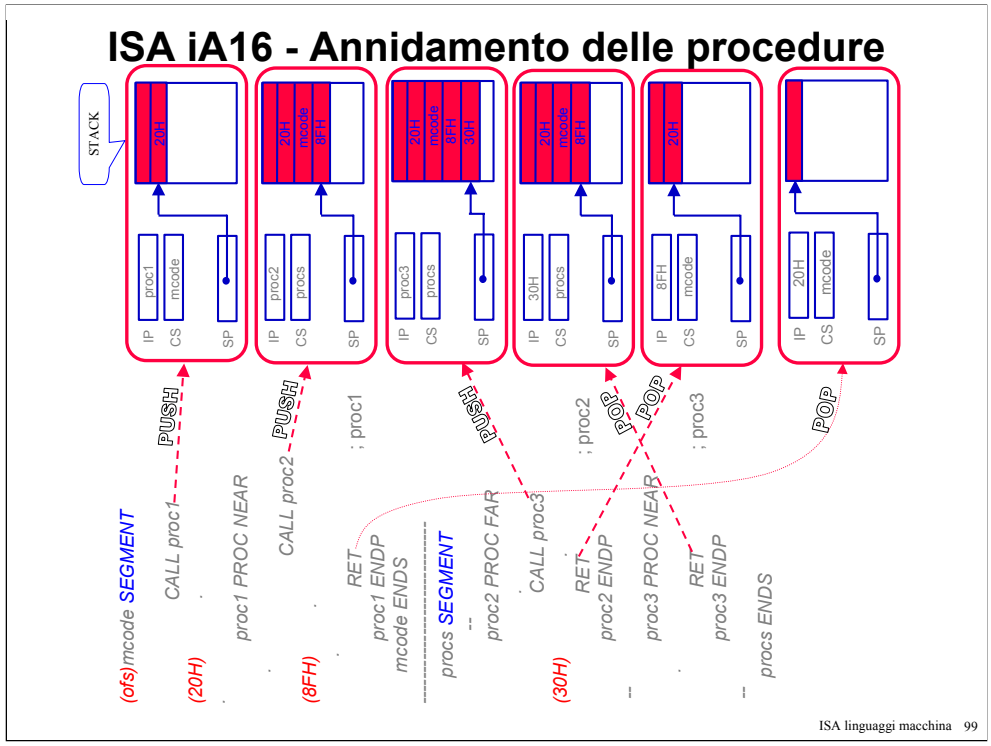
RET di tipo near

```
code1 SEGMENT
.
CALL proc1
.
proc1 PROC NEAR
.
RET                POP IP
proc1 ENDP
.
code1 ENDS
```

RET di tipo far

```
code1 SEGMENT
.
CALL proc1
.
code1 ENDS
.
code2 SEGMENT
.
proc1 PROC FAR
.
RET                POP IP, POP CS
proc1 ENDP
.
code2 ENDS
```

L'ASSEMBLER DETERMINA IL TIPO DELL'ISTRUZIONE DI RITORNO DAL TIPO DELLA LABEL UTILIZZATA NELLA DIRETTIVA "PROC".OVVIAMENTE IL TIPO DELLA CHIAMATA ED IL TIPO DEL RITORNO DEVONO ESSERE UGUALI (ATTENZIONE NELLE CALL INDIRETTE!).


ISA iA16 - Annidamento delle procedure
STACK
20H
proc1
mcode
IP
CS
SP
20H
mcode
8FH
proc2
procs
20H
mcode
8FH
30H
proc3
procs
20H
mcode
8FH
30H
procs
8FH
mcode
20H
mcode
PUSH
PUSH
PUSH
POP
POP
POP
(ofs) mcode SEGMENT
CALL proc1
(20H)
proc1 PROC NEAR
CALL proc2
(8FH)
RET
proc1 ENDP
mcode ENDS
procs SEGMENT
proc2 PROC FAR
CALL proc3
(30H)
RET
proc2 ENDP
proc3 PROC NEAR
RET
proc3 ENDP
procs ENDS
; proc1
; proc2
; proc3
ISA linguaggi macchina 99

ISA iA16 - Classi di Interruzioni

INTERRUZIONI HARDWARE

- attivate da dispositivi esterni mediante il segnale INT
- il tipo e' letto dal processore tramite un ciclo di bus (INTA)
- mascherabili (se IF=0 l'interrupt non viene servito)
- asincrone rispetto al programma in esecuzione

INTERRUZIONI SOFTWARE

- attivate mediante l'istruzione INT
- il tipo e' contenuto nell'istruzione
- non mascherabili (servite anche se IF=0)
- sincrone rispetto al programma

ECCEZIONI

- attivate al verificarsi di alcuni eventi "interni"
- il tipo e' implicito (corrispondenza 1:1 fra evento e tipo)
- non mascherabili (servite anche se IF=0)
- sincrone rispetto al programma

OVERFLOW DI DIV E IDIV → *INT 0*

OPCODE IGNOTO → *INT 6*

ALTRI TIPI RISERVATI DELL'8086:

SINGLE STEP → INT 1 (eccez. mascherabile mediante TF)

NMI → INT 2 (int. hardware non mascherabile)

BREAKPOINT → INT 3 (int. software da un byte)

ISA iA16 - Interrupt software

INT int-type

L'ISTRUZIONE INT CONSENTE DI INVOCARE UNA PROCEDURA (tipicamente "di sistema") MEDIANTE UN MECCANISMO ALTERNATIVO A QUELLO DELLE CALL. IL MECCANISMO E' QUELLO DELLE INTERRUZIONI, IL
PROCESSORE SERVE IL TIPO DI INTERRUZIONE SPECIFICATO NELL'ISTRUZIONE:

PUSH FLAG
PUSH CS
PUSH IP
IP ← *M[int-type*4]*
CS ← *M[int-type*4*+2]

LA PROCEDURA E' INVOCATA IN MODO ANONIMO: NON E' NECESSARIA L'OPERAZIONE DI COLLEGAMENTO DELLA PROCEDURA CON IL PROGRAMMA CHIAMANTE.

IRET

L'ISTRUZIONE IRET EFFETTUA IL RITORNO DA UNA PROCEDURA ATTIVATA MEDIANTE IL MECCANISMO DELLE INTERRUZIONI (sia nel caso di interruzione hardware che in quello di interruzione software):
POP IP
POP CS
POP FLAG

ISA iA16 - Abilitazione INT hardware

STI CLI

IL PROGRAMMATORE PUO' ABILITARE-DISABILITARE IL SERVIZIO DELLE INTERRUZIONI HARDWARE MANIPOLANDO IL BIT IF DEL REGISTRO DEI FLAG:

ISTRUZIONE **STI** (SET INTERRUPT FLAG) → **IF=1**

ISTRUZIONE **CLI** (CLEAR INTERRUPT FLAG) → **IF=0**

IMPIEGO DI STI
UNA PROCEDURA CHE SERVE UN INTERRUPT PARTE SEMPRE CON IF=0;
PER POTER ESSERE INTERROTTA DA UN INTERRUPT HW DEVE PORRE IF=1
(interrupt sw interrompibile da un interrupt hw, corretto annidamento degli interrupt hw)

IMPIEGO DI CLI
PROTEZIONE DI PORZIONI CRITICHE DI CODICE (es.: aggiornamento del vettore delle interruzioni)

Calcolatori Elettronici L- A

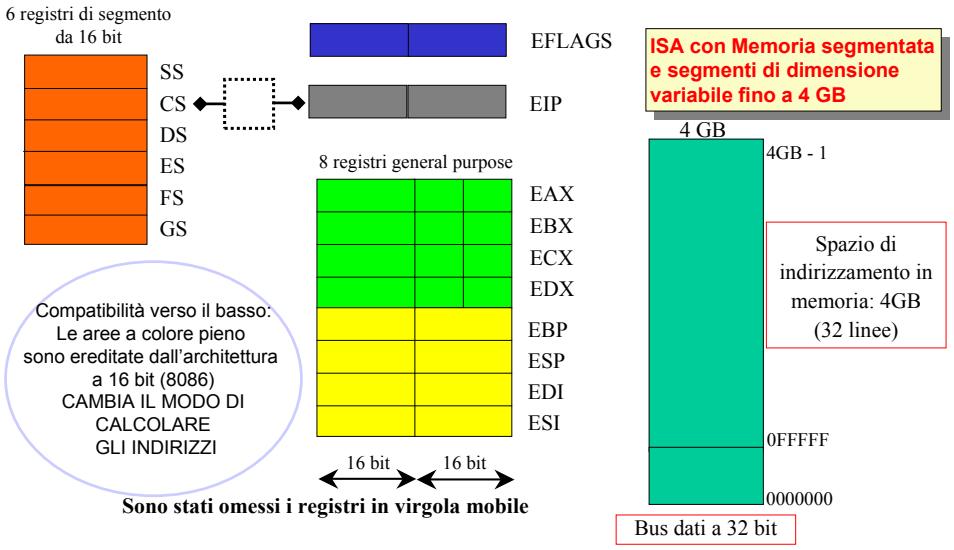
L'ISA dell'iA-32

ISA iA 32

L’acronimo iA32 proviene da Architettura intel con bus dati a 32 bit.

Storicamente succede agli ISA iA16 con l’ovvio scopo di migliorarne le prestazioni e di mantenere una compatibilità con il codice scritto per la generazione precedente; tanto è vero che il linguaggio macchina delle CPU da 32 bit e' un superset di quello dei processori i8086/88. Di essa fanno parte tutti i processori della famiglia Pentium

Ambiente di esecuzione di una applicazione nell'architettura intel a 32 bit iA32, i386, i486, Pentium, P6



Sono stati omessi i registri in virgola mobile

Architettura Intel - (Memory-Register)
Linguaggio macchina
Formato delle istruzioni (processori con architettura iA32)

Prefisso per l'istruzione (REP, REPNZ, LOCK..)	Prefisso per invertire Address-size		Prefisso per invertire Operand-size	Segment override prefix
0/1 byte	0/1 byte		0/1 byte	0/1 byte
Codice operativo bit w=1: lung. oper. è l'*operand-size* bit d=1: reg è l'operando destinazione	**MOD r/m** mod *(7,6)* reg/**cop** ***(5,4,3)*** r/m *(2,1,0)*	**S.I.B. byte** Scale *(7,6)* Index *(5,4,3)* Base *(2,1,0)*	Displacement	Operando immediato
1/2 byte	0/1 byte	0/1 byte	0/1/2/4 byte	0/1/2/4 byte

- Con address size = 16 bit ⇒ il byte S.I.B. (è un record a tre campi: Scale, Index, Base) è assente e sono ammesse solo le modalità di indirizzamento dell'8086 (con la stessa codifica)
- Con address size = 32 ⇒ cambia l'interpretazione di r/m per consentire l'ortogonalità tra registri e modalità di indirizzamento:
  - r/m = 100 ⇒ S.I.B. è presente, *base* e *index* (3 bit ciascuno) identificano i registri base e indice di 32 bit e SS codifica il fattore di scala per *Index*
  - mod=10 ⇒ il displacement è di 32 bit, altrimenti può essere assente (mod=00), oppure di 8 bit (mod=01)

Esempio di codice in assembler dell'architettura Intel iA32

Si scriva il codice assembler per il calcolo della somma degli elementi di un vettore A di 8 elementi

```
Codice segment er use32                ;er significa Execute + Read
        Inizializzazione di DS e SS
        mov     ecx,    8              ; inizializzazione del contatore
        mov     esi,    0              ; inizializzazione dell'indice
        mov     eax,    0              ; inizializzazione della somma corrente

CICLO   add     eax,  A[esi*4]         ; sull'indice si applica il fattore di scala pari alla
                                       ; dimensione degli elementi del vettore
        inc     ESI
        loop    CICLO
        mov     Z, eax                 ; scrivi il risultato in Z
        ……
Codice ends

Dati    segment rw                     ; rw significa Read + Write
A       dd      8 dup (?)
Z       dd      ?
Dati    ends
```

Esercizi sul L.M. dell'architettura iA32

- Con riferimento al lucido precedente si eseguano i seguenti esercizi:
  - si indichi quali istruzioni fanno riferimento alla memoria e quali no
  - si dica a quale categoria appartiene ciascuna delle istruzioni
  - si disegni lo spazio di indirizzamento in memoria della macchina iA32 e si posizioni all'interno di esso i due segmenti di codice e dati
  - Si disegni la dinamica del PC durante l'esecuzione del programma
  - Si elenchino i valori che eax assume durante l'esecuzione del programma
  - Si calcoli il numero di cicli di bus che la CPU esegue durante l'esecuzione del programma suddividendoli in cicli di bus di FETCH e cicli di bus di EXECUTE

Linguaggio macchina nell'architettura Intel (Memory-Register) Compatibilita' tra macchine da 16 e 32 bit (iA16 iA32)

- il linguaggio macchina delle CPU da 32 bit e' un superset di quello dei processori i8086/88
- Nell'architettura a 32 bit (iA32) ad ogni segmento è associato un descrittore di segmento
- nei descrittori dei segmenti di codice delle macchine da 32 bit e' presente un bit (detto D) che stabilisce se il codice associato e' da 16 o 32 bit
- In questo modo si realizza la compatibilità della nuova architettura con la vecchia (compatibilità verso il basso)

Linguaggio macchina nell'architettura Intel (Memory-Register)
Compatibilita' tra macchine da 16 e 32 bit
Codice di iA16 eseguito in iA32

- Se nel descrittore di un segmento di codice si ha bit D = 0, allora il contenuto del segmento è interpretato come codice di iA16. In questo caso:
  - gli Effective Address (cioe' gli offset nei segmenti) e i registri general purpose sono considerati da 16 bit
  - il formato delle istruzioni e' quello dell'8086
- codice compilato per macchine a 16 bit puo' essere eseguito da macchine a 32 bit inserendo detto codice in segmenti con D=0

Linguaggio macchina nell'architettura Intel (Memory-Register)
Codice eseguibile in iA32

- Nel caso del codice da 32 bit (D = 1) cambia rispetto al caso precedente l'interpretazione del codice eseguibile:
  - gli offset nei segmenti e i registri general purpose sono considerati da 32 bit
  - il formato delle istruzioni e' modificato per accettare altre modalita' di indirizzamento (es. viene introdotto un fattore di scala sugli indici e si ha la quasi completa ortogonalita' tra modalita' di indirizzamento e registri base/indice)

Linguaggio macchina nell'architettura Intel (Memory-Register)
Codice da 16 e 32 bit nello stesso segmento

- E' possibile mescolare nello stesso segmento codice a 16 e a 32 bit premettendo alle istruzioni "deviate" (cioe' da 16 bit in codice a 32 o da 32 bit in codice a 16) uno o due opportuni prefissi che invertono (sulle sole istruzioni col prefisso) la modalita' di interpretazione dell'operando e/o della modalita' di indirizzamento
  - Es. 1: l'istruzione INC AX puo' essere inserita in un codice a 32 bit, ma l'istruzione macchina dovra' avere un prefisso detto "operand-size prefix"
  - Es. 2: l'istruzione INC EAX puo' essere inserita in un codice a 16 bit, ma l'istruzione macchina dovra' avere l' "operand-size prefix"

Linguaggio macchina nell'architettura Intel iA32
(Memory-Register) Segmentazione della memoria (1)

- Ad ogni segmento e' associato un descrittore di 8 byte contenente tutti gli attributi del segmento (indirizzo di origine, lunghezza, diritti di accesso, tipo, etc.)
- I descrittori sono riuniti in apposite tabelle dette "tabelle dei descrittori" (Global Descriptor Table - GDT e Local Descriptor Table (LDT)
- e' possibile accedere a un descrittore attraverso una chiave di accesso di 16 bit (detta selettore) contenente sia l'identificatore della tabella (GDT o LDT) sia l'indice del descrittore nella tabella (13 bit)
- Per accedere a un oggetto in memoria e' necessario conoscere il segmento di appartenenza; il relativo selettore deve essere caricato in un registro di segmento
- i registri di segmento sono sei, quindi in ogni istante il programma ha la visibilita' di non piu' di sei segmenti

Linguaggio macchina nell'architettura Intel iA32
(Memory-Register) Segmentazione della memoria (2)

- L'architettura Intel a 32 bit e' chiamata anche:
  - architettura a 32 bit multitasking protetta
- La segmentazione della memoria e l'associazione a ogni segmento del relativo descrittore e la suddivisione dei descrittori in due tabelle (locale e globale) costituiscono il supporto di base dell'architettura dell'hardware alle seguenti funzionalità:
  - il controllo sugli accessi in memoria (protezione)
  - la definizione di processi e delle risorse di memoria ad essi associati
  - la separazione tra processi (protezione)
  - la commutazione tra un processo e l'altro (task switching)

Misure sulla frequenza di esecuzione delle istruzioni al variare dell'architettura

Architettura	M - R (2-1)	R - R (3-0)	M - M (3-3)
Tipo di Istruzioni	Es.: 8086	Es.: DLX	Es.: VAX
Control transfer	24 % 14% cond branch	15 % 11% cond branch	28 % 17% cond branch
Data transfer	42 % 27% mov	30 % 22% LD e 8% ST	19 %
Alu	30 %	51 % 20% add	48 %

- Dati indicativi mediati su molti benchmark non tutti coincidenti
- Il rapporto tra istruzioni control transfer è una stima del rapporto tra le densità di codice
- Detto rapporto dipende sia dal L.M. sia dall'efficienza del compilatore