

IL LIVELLO DI MACROARCHITETTURA (ISA) (Seconda Parte)

**IL LIVELLO DEL LINGUAGGIO
ASSEMBLATIVO**

Argomenti

- IL LIVELLO DI MICROARCHITETTURA
 - Modalità di indirizzamento
 - Tipi di istruzioni
 - Controllo del flusso
 - Architetture Intel IA-32 e IA64
- IL LIVELLO DEL LINGUAGGIO ASSEMBLATIVO
 - Introduzione al linguaggio assemblativo
 - Formato delle istruzioni
 - Pseudoinstruzioni
- LE MACROISTRUZIONI
 - Macro vs procedure
 - Macro con parametri
- IL PROCESSO DI ASSEMBLAGGIO
 - Primo e secondo passaggio
- LINKER E LOADER

Modalità di Indirizzamento

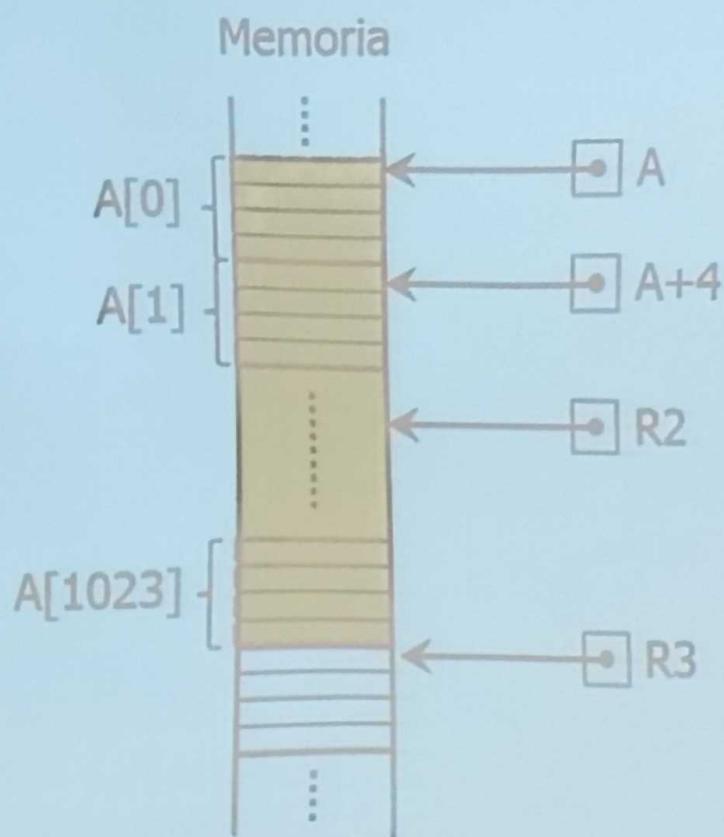
- **Immediato:** il valore dell'operando è nell'istruzione.
- **Diretto:** l'istruzione contiene l'indirizzo di memoria completo dell'operando (variabili globali).
- **Indiretto:** l'indirizzo di memoria fornito contiene l'indirizzo dell'operando.
- **A registro:** si specifica un registro che contiene l'operando.

Modalità di Indirizzamento

- **Indiretto a registro:** il registro specificato contiene l'indirizzo dell'operando.
- **Indicizzato:** l'indirizzo è dato da una costante più il contenuto di un registro.
- **A registro base:** viene sommato a tutti gli indirizzi il contenuto di un registro.
- **A stack:** l'operando è sulla cima dello stack (o ci deve andare).

Esempio

- Programma che calcola la somma degli elementi di un array (1024 interi) che iniziano all'indirizzo A.



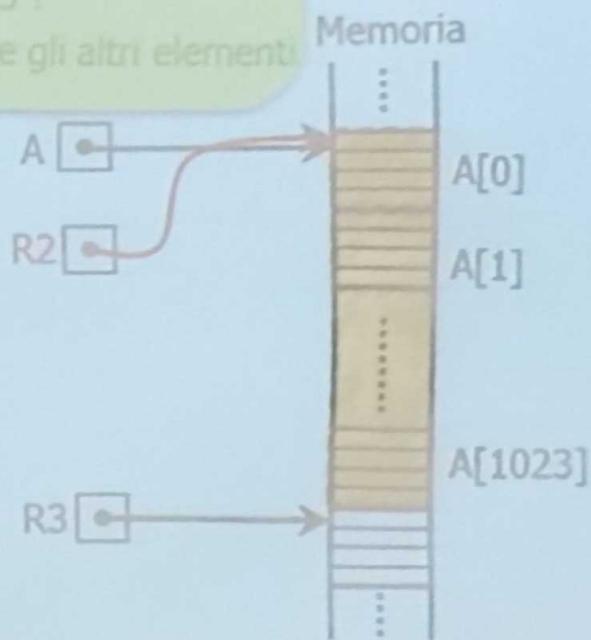
- un intero occupa 4 byte.
- la somma viene accumulata in R1.
- R2 punta all'elemento corrente.
- R3 è la prima posizione esterna all'array.

R1

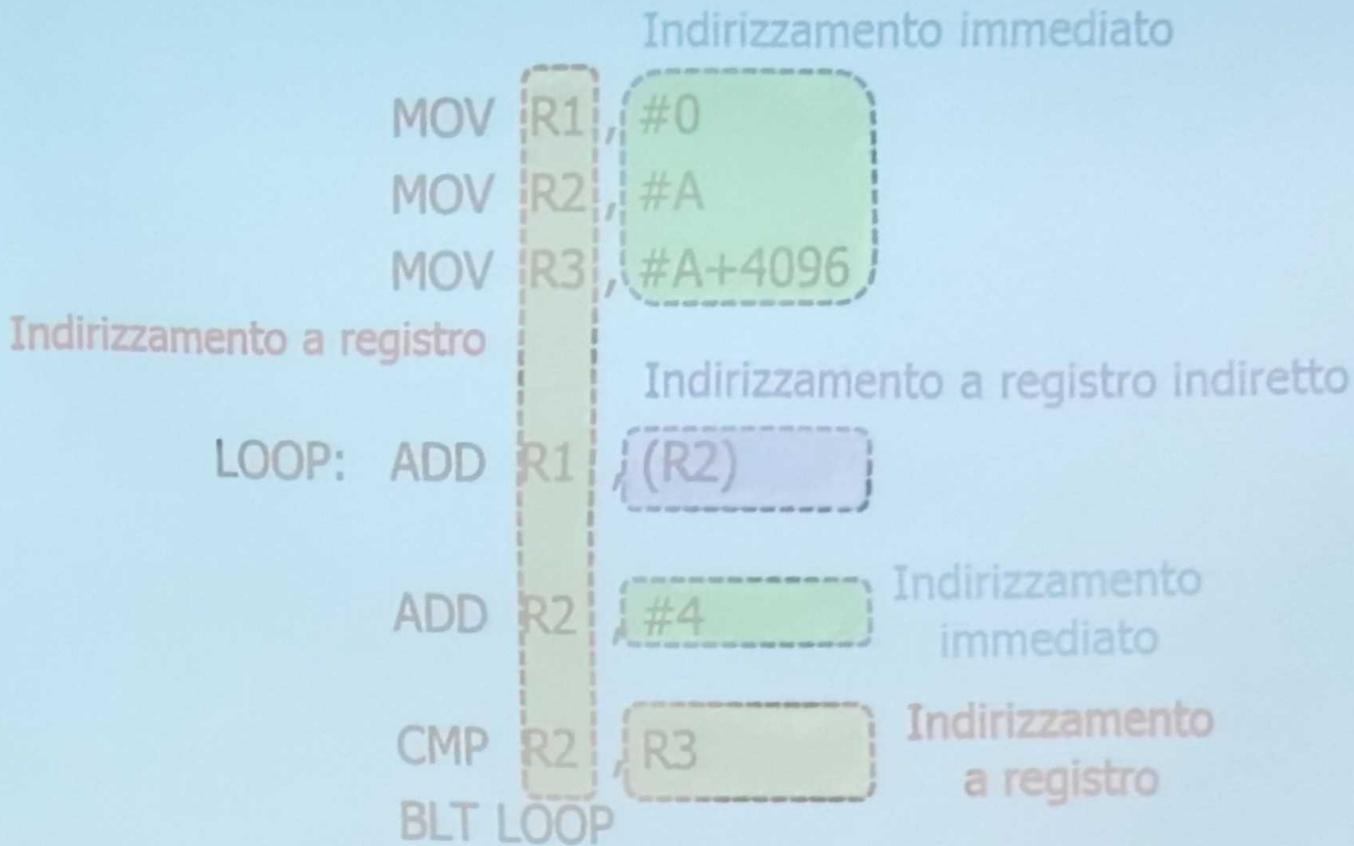
0

Esempio di indirizzamento a registro indiretto

```
MOV R1,#0      ; R1 registra la somma parziale, inizialmente 0  
MOV R2,#A      ; in R2 c'è l'indirizzo dell'array A  
MOV R3,#A+4096 ; in R3 c'è l'indirizzo oltre l'array A  
LOOP: ADD R1,(R2)    ; R1=R1+ (il valore puntato da R2)  
        ADD R2,#4      ; incrementa R2 di una parola (4 byte)  
        CMP R2,R3      ; R2 ha raggiunto l'indirizzo R3 ?  
        BLT LOOP       ; se R2 < R3 occorre sommare gli altri elementi
```



Istruzioni al microscopio



Esempio di codice auto-modificante

- Questo programma utilizza una istruzione che si modifica durante l'esecuzione:

```
MOV R1,#0          ; accumula la somma in R1, inizialmente a 0
MOV R2,#4          ; R2 = 4
MOV R3,#A+4096    ; R3 = indirizzo prima parola dopo array A
LOOP: ADD R1,A+R2  ; somma ad A un offset crescente
       ADD R2,#4      ; incrementa R2 di una parola (4 byte)
       CMP R2,R3      ; R2==R3 ?
       BLT LOOP        ; se R2 < R3 si devono sommare altri elementi
```

- L'idea fu di Von Neumann quando non esisteva l'indirizzamento indiretto a registro

Indirizzamento indicizzato

- L'esempio calcola l'AND bit a bit di due array A e B e pone in OR tutti i risultati parziali:

$(A[0] \text{ AND } B[0]) \text{ OR } (A[1] \text{ AND } B[1]) \text{ OR } (A[2] \text{ AND } B[2])$
OR ...

- R1 accumula l'OR degli AND.
- R2 indica la posizione corrente sugli array.

Indirizzamento indicizzato

- R3 contiene la costante 4096, per controllare la fine del loop.
- R4 è utilizzato per calcolare i singoli AND.

```
MOV R1,#0      ; accumula in R1 l'OR, inizialmente 0
MOV R2,#0      ; R2 = posizione corrente nei due array A e B
MOV R3,#4096   ; R3 = first index value not to use
LOOP:    MOV R4,A(R2)    ; R4 = A[R2]
          AND R4,B(R2)    ; R4 = A[R2] AND B[R2]
          OR R1,R4       ; R1 = R1 OR R4
          ADD R2,#4       ; R2 = R2 + 4
          CMP R2,R3      ; R2==R3 ?
          BLT LOOP        ; se R2 < R3 occorre considerare le altre celle
```

Indirizzamento indicizzato esteso

- L'indirizzo di memoria è calcolato sommando il contenuto di due registri:
 - Un registro memorizza la base.
 - Un registro memorizza l'indice.
- Ad esempio inizializzando R5 con A e R6 con B:

LOOP:

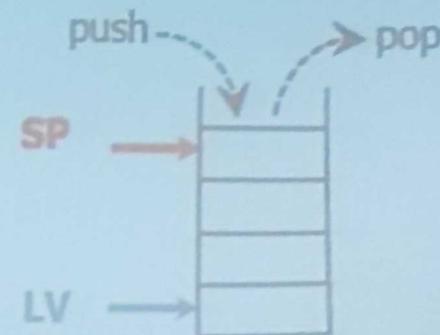
...
MOV R4, (R2+R5)

AND R4, (R2+R6)

...

Indirizzamento a stack

- Lo **Stack** è utilizzato per:
 - Gestire le chiamate di procedura.
 - Calcolare espressioni aritmetiche.
 - Salvare risultati intermedi.
- Lo Stack Pointer **SP** punta all'elemento affiorante della struttura.
- Le operazioni principali:
 - **push**: aggiunge un elemento alla cima dello stack.
 - **pop**: preleva un elemento dalla cima dello stack.
 - Operazioni aritmetiche sugli elementi affioranti (l'elemento al top è l'operando di destra mentre l'altro quello di sinistra).



Notazione Polacca Postfissa (o Inversa)

- La notazione polacca inversa (**RPN**) fu chiamata così in analogia alla notazione polacca (inventata da Łukasiewicz).
- È un modo per esprimere le espressioni matematiche in modo implicito senza l'utilizzo delle parentesi:

$$(A+B) \times C \rightarrow AB+CX$$

Forma infissa	Forma postfissa
$A + B \times C$	$ABC\times+$
$A \times B + C$	$AB\times C+$
$A \times B + C \times D$	$AB\times CD\times+$
$(A + B) / (C - D)$	$AB+CD-/$
$A \times B / C$	$AB\times C/$
$((A + B) \times C + D) / (E + F + G)$	$AB+ C\times D+ EF+ G+/$

RPN ed esecuzione

- Supponiamo che la JVM debba valutare l'espressione:

$$(8 + 2 \times 5) / (1 + 3 \times 2 - 4)$$

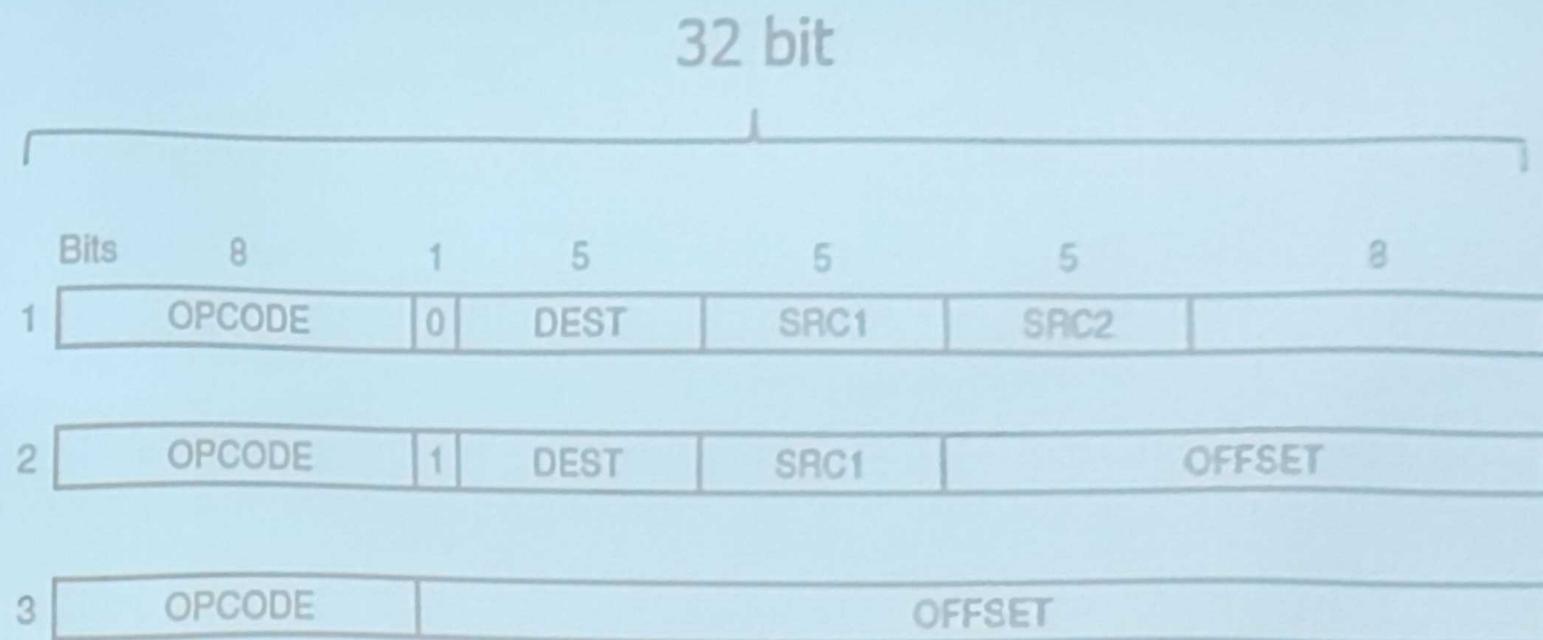
- che scritta in RPN risulta: 8 2 5 × + 1 3 2 × + 4 - /

Step	Remaining string	Instruction	Stack
1	8 2 5 × + 1 3 2 × + 4 - /	BIPUSH 8	8
2	2 5 × + 1 3 2 × + 4 - /	BIPUSH 2	8, 2
3	5 × + 1 3 2 × + 4 - /	BIPUSH 5	8, 2, 5
4	× + 1 3 2 × + 4 - /	IMUL	8, 10
5	+ 1 3 2 × + 4 - /	IADD	18
6	1 3 2 × + 4 - /	BIPUSH 1	18, 1
7	3 2 × + 4 - /	BIPUSH 3	18, 1, 3
8	2 × + 4 - /	BIPUSH 2	18, 1, 3, 2
9	× + 4 - /	IMUL	18, 1, 6
10	+ 4 - /	IADD	18, 7
11	4 - /	BIPUSH 4	18, 7, 4
12	- /	ISUB	18, 3
13	/	IDIV	6

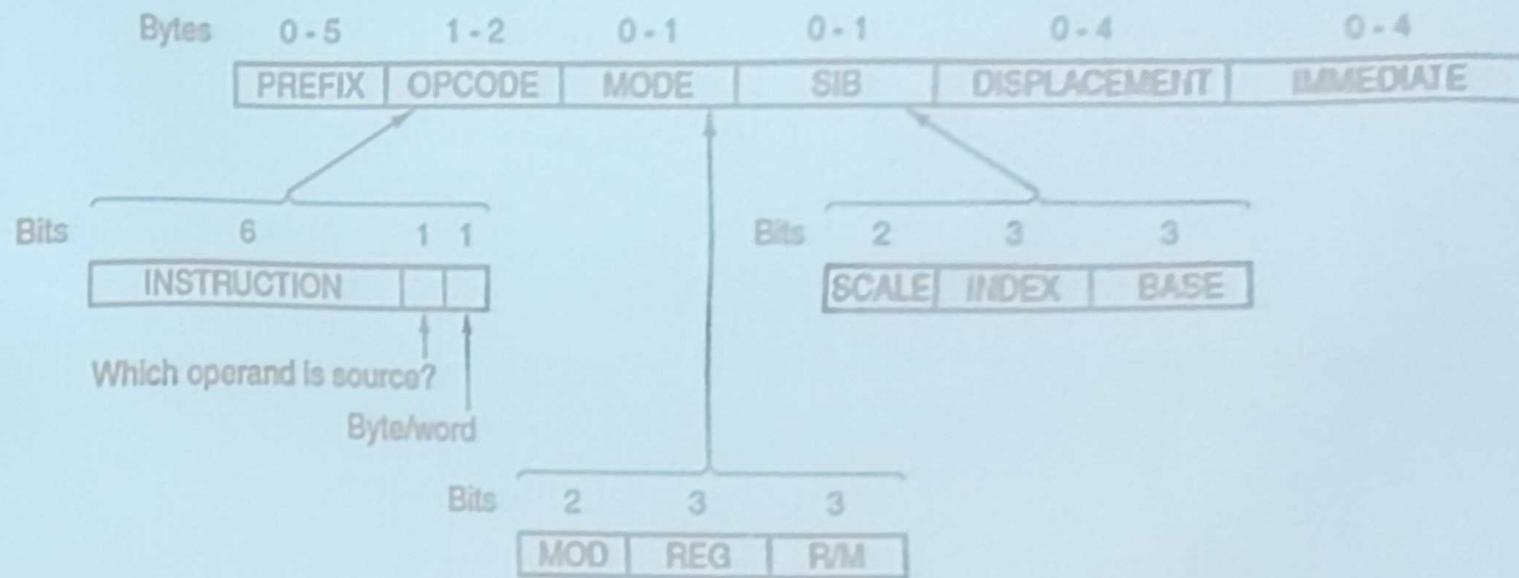
Ortogonalità

- Un set di istruzioni è caratterizzato da:
 - Codici operativi.
 - Modalità di indirizzamento.
- Quando tutte modalità di indirizzamento sono utilizzabili con tutti i codici operativi si dice che i codici e le modalità di indirizzamento sono tra loro **ortogonali**.
- Sebbene l'ortogonalità sia una caratteristica desiderabile perché semplifica la generazione del codice (es. SPARC) le architetture Intel non hanno questa caratteristica.

Instruction Set ortogonale



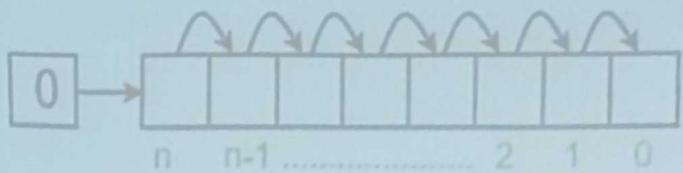
Instruction Set non ortogonale



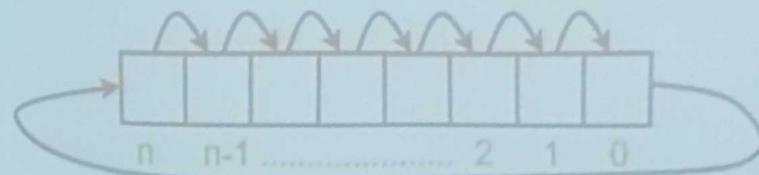
Tipi di istruzioni

- **Unarie**
 - Aritmetiche: complemento, opposto, incremento, radice,...
 - Bit a bit: not, shift e rotation.
- **Binarie**
 - Aritmetiche e logiche.
- **Trasferimento dati**
 - Da registro/memoria a registro/memoria (4 casi).
- **Selezione e confronto**
- **Iterazione**
- **Chiamata di una procedura**
- **Input/Output**
 - Polling, interrupt, DMA.

Istruzioni unarie sui bit



Shift a destra

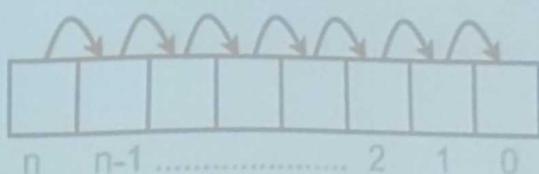


Rotazione a destra

00000000 00000000 00000000 01110011 A

00000000 00000000 00000000 00011100 Doppio shift a destra di A

11000000 00000000 00000000 00011100 Doppia rotazione a destra di A



Shift a destra con estensione del segno

1111111 1111111 1111111 11110000 A

00111111 1111111 1111111 11111100 Shift di A senza estensione del segno

1111111 1111111 1111111 11111100 Shift di A con estensione del segno

Applicazione dello shift

- Scorrere di k bit a sinistra/destra un **numero intero** X significa moltiplicarlo/dividerlo per 2^k (a meno di overflow/underflow).
- In base alla proprietà distributiva, si può estendere il ragionamento e calcolare rapidamente operazioni apparentemente complesse:

$$18 \cdot n = 2^4 \cdot n + 2^1 \cdot n$$

- In questo caso invece che eseguire una moltiplicazione (necessaria poiché non abbiamo una potenza di due) si può eseguire una somma e 5 scorrimenti:
 - $2^4 \cdot n$  corrisponde a scorrere n di 4 bit a sinistra
 - $2^1 \cdot n$  corrisponde a scorrere n di 1 bit a sinistra
 - Si calcola la somma dei due risultati

Attenzione alle istruzioni di shift

- In generale lo shift a destra può introdurre degli errori a causa del troncamento delle cifre che compongono il risultato
- Invece nel caso dei numeri negativi lo shift a sinistra non produce la moltiplicazione per 2 (contrariamente allo shift a destra che, troncamenti a parte, funziona correttamente)

Altre istruzioni unarie

- **CLR**, azzeramento.
- **INC**, incremento.
- **NEG**, complemento alla base (complemento a uno più uno)
$$X + \text{NEG}(X) = 0$$
- **NOT**, negazione.
- **SQRT**, radice quadrata.

Operazioni binarie

- **Logiche**
 - AND, OR, XOR, NAND, NOR.
- **Aritmetiche**
 - ADD, MULT, DIV,...

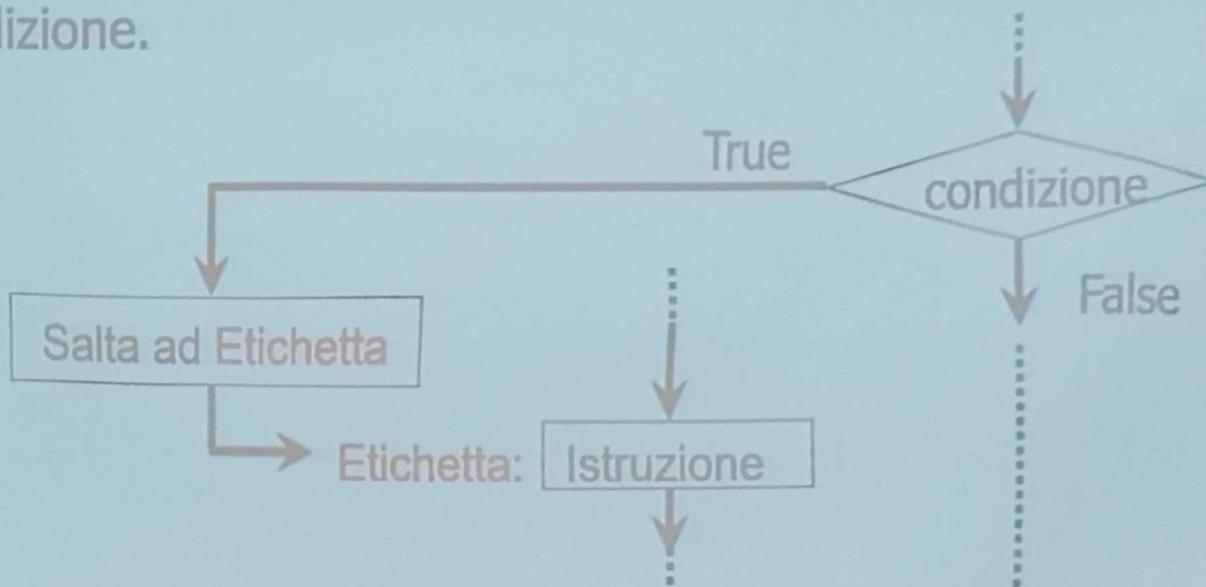
Non si confonda il termine binario che nel caso delle operazioni si riferisce al fatto che ci sono solo due operandi

Esempi

10110111 10111100 11011011 10001011 A
11111111 11111111 11111111 00000000 B (mask)
10110111 10111100 11011011 00000000 A AND B
00000000 00000000 00000000 01010111 C
10110111 10111100 11011011 01010111 (A AND B) OR C

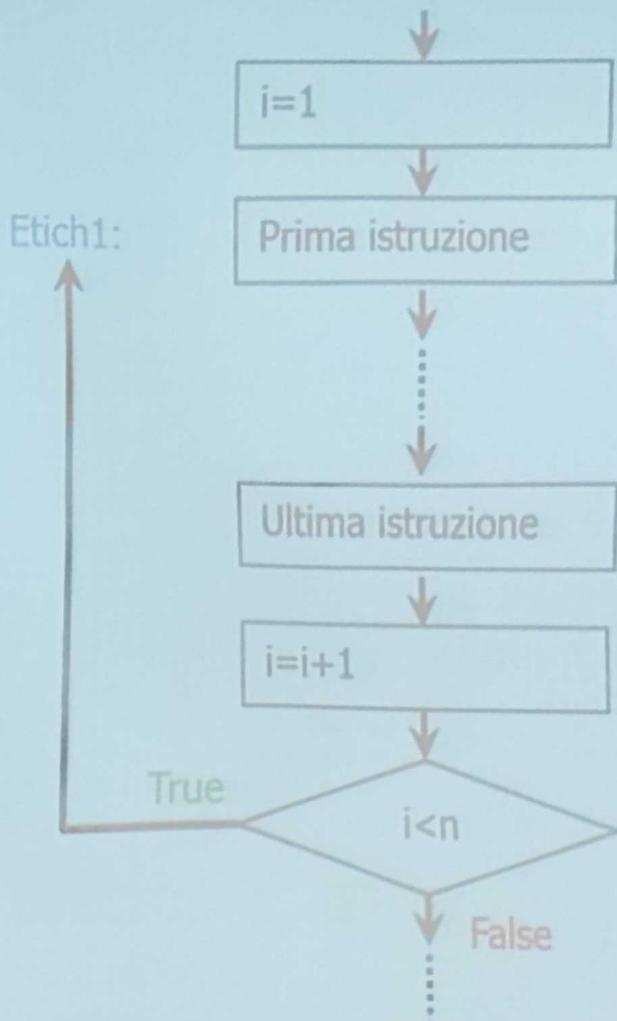
Istruzioni di selezione

- Un modo comune per eseguire la selezione nel flusso di esecuzione di un algoritmo è attraverso la verifica di una condizione.



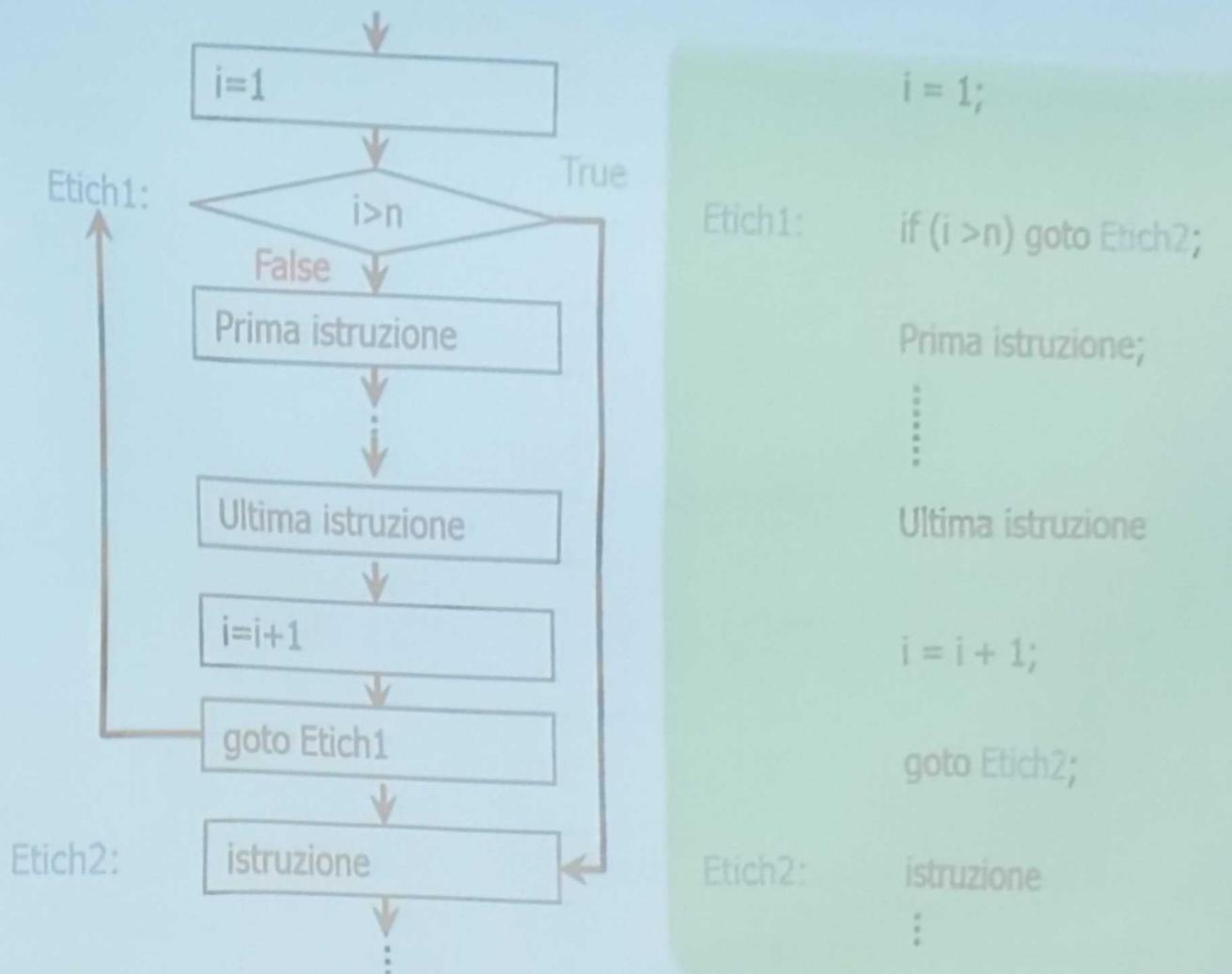
- La verifica della condizione normalmente si effettua attraverso la PSW.

Istruzione di iterazione: ciclo do..while



```
i = 1;  
Etich1: Prima istruzione;  
...  
Ultima istruzione  
  
i = i + 1;  
  
if (i < n) goto Etich1;  
...
```

Istruzione di iterazione: ciclo while..do



Istruzioni di I/O

- Un'operazione di I/O consiste nel trasferimento di dati verso/da un dispositivo periferico.
- Esistono tre modi per gestire l'I/O:

1) I/O programmato con busy waiting (attesa attiva)

La CPU interroga periodicamente i dispositivi (polling) rimanendo in attesa (busy waiting) che il dispositivo sia pronto al trasferimento.

2) I/O gestito con interruzioni

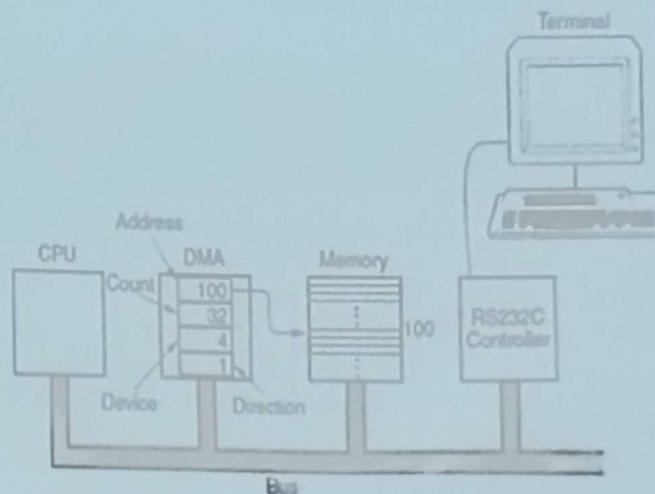
La CPU non si preoccupa dei dispositivi poiché sono loro che devono richiedere il trasferimento dati attraverso l'interruzione.

3) DMA (Direct Memory Access)

La CPU avvia l'operazione che viene poi gestita interamente dal controllore DMA.

Direct Memory Access

- La CPU programma il controller DMA specificando:
 - Dimensione del trasferimento (byte).
 - Il dispositivo da interrogare.
 - In quale zona di memoria (indirizzi).
- Il controller gestisce l'intera operazione.
- Il DMA può gestire più operazioni contemporaneamente.



Istruzioni Intel Core i7 (1/2)

Moves

MOV DST,SRC	Move SRC to DST
PUSH SRC	Push SRC onto the stack
POP DST	Pop a word from the stack to DST
XCHG DS1,DS2	Exchange DS1 and DS2
LEA DST,SRC	Load effective addr of SRC into DST
CMOVcc DST,SRC	Conditional move

Boolean

AND DST,SRC	Boolean AND SRC into DST
OR DST,SRC	Boolean OR SRC into DST
XOR DST,SRC	Boolean Exclusive OR SRC to DST
NOT DST	Replace DST with 1's complement

Arithmetic

ADD DST,SRC	Add SRC to DST
SUB DST,SRC	Subtract SRC from DST
MUL SRC	Multiply EAX by SRC (unsigned)
IMUL SRC	Multiply EAX by SRC (signed)
DIV SRC	Divide EDX:EAX by SRC (unsigned)
IDIV SRC	Divide EDX:EAX by SRC (signed)
ADC DST,SRC	Add SRC to DST, then add carry bit
SBB DST,SRC	Subtract SRC & carry from DST
INC DST	Add 1 to DST
DEC DST	Subtract 1 from DST
NEG DST	Negate DST (subtract it from 0)

Binary coded decimal

DAA	Decimal adjust
DAS	Decimal adjust for subtraction
AAA	ASCII adjust for addition
AAS	ASCII adjust for subtraction
AAM	ASCII adjust for multiplication
AAD	ASCII adjust for division

Shift/rotate

SAL/SAR DST,#	Shift DST left/right # bits
SHL/SHR DST,#	Logical shift DST left/right # bits
ROL/ROR DST,#	Rotate DST left/right # bits
RCL/RCR DST,#	Rotate DST through carry # bits

Test/compare

TEST SRC1,SRC2	Boolean AND operands, set flags
CMP SRC1,SRC2	Set flags based on SRC1 - SRC2

SRC = source

DST = destination

= shift/rotate count

LV = # locals

Istruzioni Intel Core i7 (2/2)

Transfer of control

JMP ADDR	Jump to ADDR
Jxx ADDR	Conditional jumps based on flags
CALL ADDR	Call procedure at ADDR
RET	Return from procedure
IRET	Return from interrupt
LOOPxx	Loop until condition met
INT n	Initiate a software interrupt
INTO	Interrupt if overflow bit is set

Strings

LODS	Load string
STOS	Store string
MOVS	Move string
CMPS	Compare two strings
SCAS	Scan Strings

SRC = source
DST = destination

= shift/rotate count
LV = # locals

Condition codes

BT	Set carry bit in EFLAGS register
CLC	Clear carry bit in EFLAGS register
CMC	Complement carry bit in EFLAGS
STD	Set direction bit in EFLAGS register
CLD	Clear direction bit in EFLAGS register
STI	Set interrupt bit in EFLAGS register
CLI	Clear interrupt bit in EFLAGS register
PUSHFD	Push EFLAGS register onto stack
POPFD	Pop EFLAGS register from stack
LAHF	Load AH from EFLAGS register
BAHF	Store AH in EFLAGS register

Miscellaneous

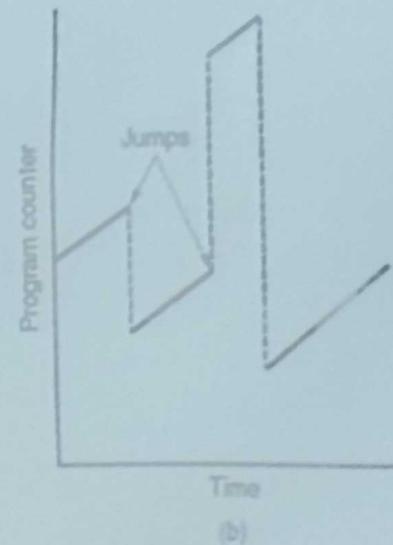
SWAP DST	Change orientation of DST
CWQ	Extend EAX to ECX/EAX by shifts
CWDE	Extend 16-bit number in AX to EAX
ENTER SIZE,LV	Create stack frame with SIZE bytes
LEAVE	Undo stack frame built by ENTER
NOP	No operation
HLT	Halt
IN AL,PORT	Input a byte from PORT to AL
OUT PORT,AL	Output a byte from AL to PORT
WAIT	Wait for an interrupt

Controllo del flusso

- Tecniche che possono alterare l'esecuzione:
 - Salti (condizionati e non).
 - Chiamata di procedure.
 - Coroutine.
 - Trap.
 - Interrupt.



(a)



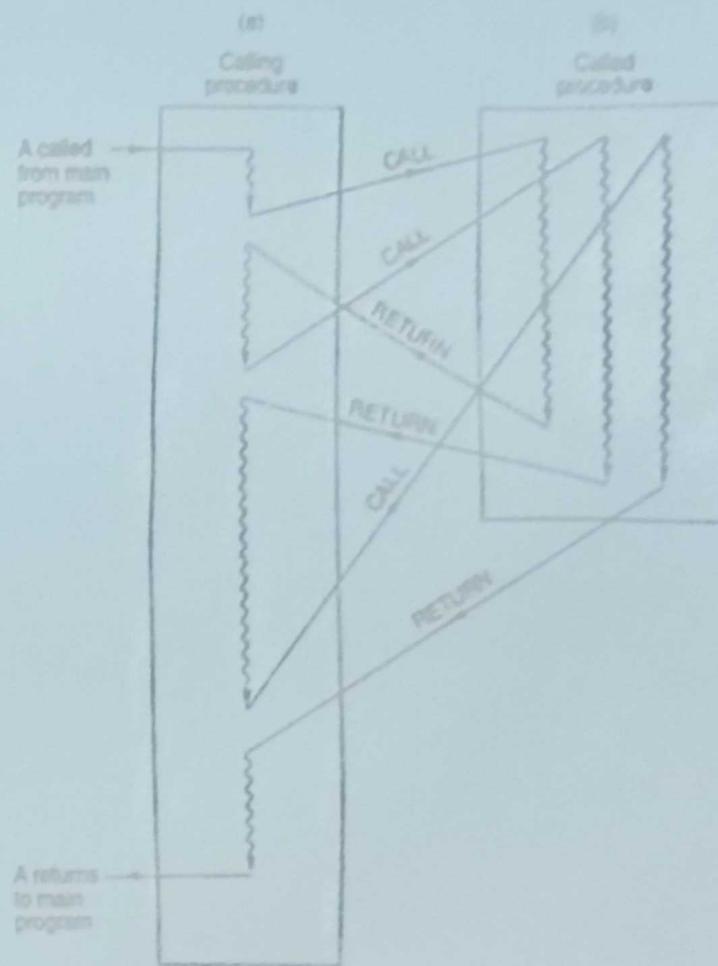
(b)

Procedure

- Per ciascuna chiamata ad una procedura viene allocato sullo stack un nuovo **stack frame** che contiene:
 - I parametri in entrata e in uscita.
 - Le variabili locali.
 - L'indirizzo di rientro.
 - Un puntatore allo stack frame del chiamante.
- Lo stack pointer **SP** punta la cima dello stack, mentre il base pointer **BP** alla base del frame.
- L'accesso ai parametri e alle variabili locali e avviene tramite offset da **BP**, poiché questo riferimento rimane costante anche se sono inseriti nuovi stackframe.
- Quanto la procedura è terminata lo stack frame viene deallocated.

Procedura

- Nelle procedure c'è una netta distinzione tra modulo chiamante (master) e modulo chiamato (slave).
- Si osservi come il chiamato (**b**) inizi sempre dalla prima istruzione mentre il chiamante (**a**) prosegua nell'esecuzione, questo fatto determina un'asimmetria.

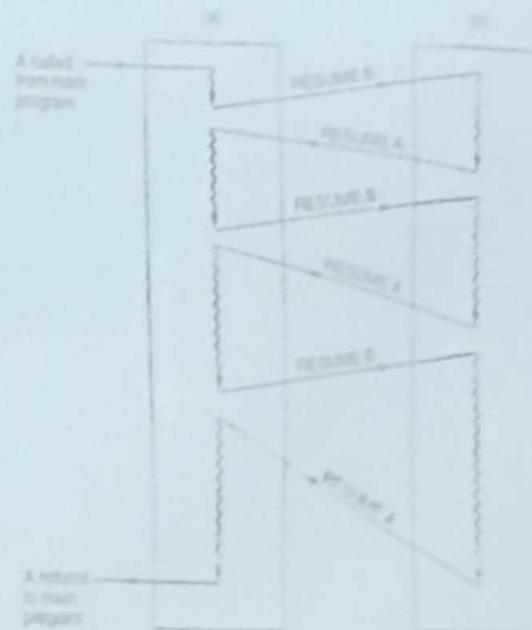


Trap

- La trap è una procedura automatica attivata da un evento eccezionale che si verifica durante l'esecuzione di un programma.
- Le trap si originano da test fatti a livello del microprogramma.
- La gestione delle trap è affidata al *trap handler*.
- **Esempi di trap:**
 - Overflow e underflow.
 - Opcode non definiti.
 - Violazione di protezione.
 - Divisione per zero.
 - Tentativi di utilizzo di dispositivi inesistenti.

Coroutine

- Nelle coroutine la situazione è completamente simmetrica.
- RESUME al posto di CALL e RETURN.
- Ciascuna RESUME riparte dall'istruzione successiva a quella precedentemente lasciata.
- Sono utilizzate per le simulazioni parallele su singola CPU.
- È utile per la validazione del software destinato a girare sui sistemi multiprocessore.



Trap

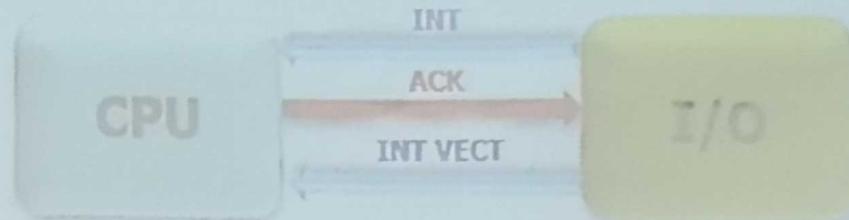
- La trap è una procedura automatica attivata da un evento eccezionale che si verifica durante l'esecuzione di un programma.
- Le trap si originano da test fatti a livello del microprogramma.
- La gestione delle trap è affidata al *trap handler*.
- **Esempi di trap:**
 - Overflow e underflow.
 - Opcode non definiti.
 - Violazione di protezione.
 - Divisione per zero.
 - Tentativi di utilizzo di dispositivi inesistenti.

Interrupt

- Gli **interrupt** sono eventi, spesso correlati con l'I/O, che cambiano il normale flusso di esecuzione.
- Mentre le trap sono sincrone e dipendenti da quello che succede all'interno della CPU, gli interrupt sono asincroni e nascono all'esterno della CPU.
- La gestione delle interruzioni è affidata *all'interrupt handler* e la routine di gestione dell'interrupt è chiamata **ISR** (Interrupt Service Routine).
- Al verificarsi dell'interrupt si intraprendono delle azioni e si esegue un certo codice, ma quando tutto è finito si deve riprendere l'esecuzione (e lo stato) dal punto esatto in cui è stata lasciata al momento in cui è accaduto l'interrupt. Questa si chiama **trasparenza** dell'interrupt.

Azioni hardware di gestione dell'interrupt

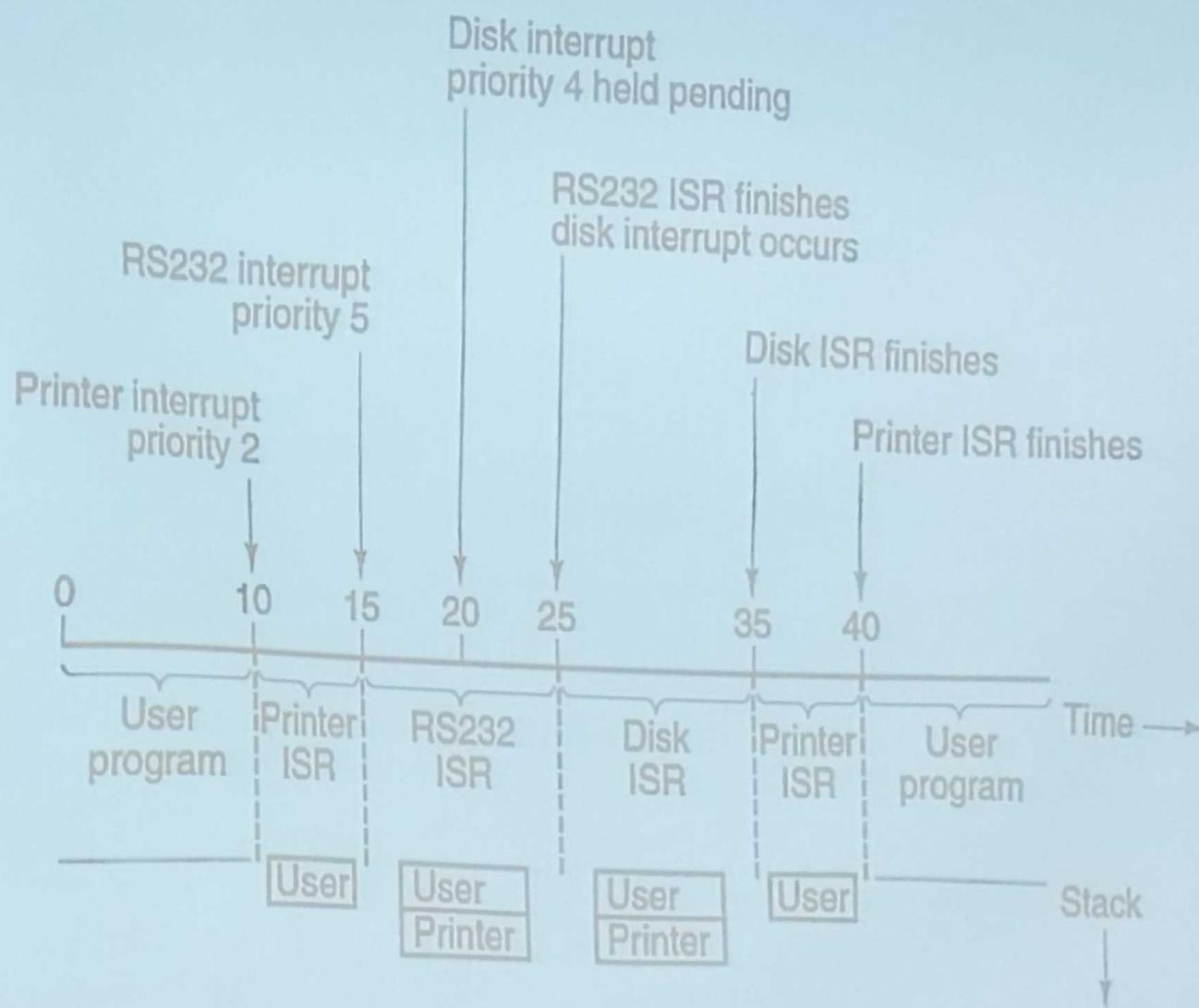
- Quando l'interruzione si origina e viene servita, vengono svolte a livello hardware queste azioni:
 - Il controller genera l'interruzione.
 - La CPU, quando è pronta a servirla, invia il segnale di **acknowledge**.
 - Quando il controller vede l'**acknowledge** risponde mettendo sul bus un identificatore detto **vettore di interruzione**.
 - La CPU legge e salva il vettore di interruzione.
 - La CPU salva il PC (Program Counter) e la PSW (Program Status word) sullo stack.
 - La CPU individua, per il tramite del vettore di interruzione, l'indirizzo iniziale della ISR e lo carica nel PC.



Azioni software di gestione dell'interrupt

- Inizia ora l'esecuzione della routine di servizio che svolge le seguenti azioni:
 1. La ISR salva sullo stack i registri della CPU per poterli ripristinare.
 2. Individua il numero esatto del device tramite lettura di opportuni registri poiché il vettore identifica solo il tipo di device che ha generato l'interruzione.
 3. Legge tutti tutte le informazioni che caratterizzano l'interruzione (es. codici di stato).
 4. Gestisce eventuali errori di I/O.
 5. Esegue ciò che è previsto per gestire l'interruzione.
 6. Se necessario informa il device che il servizio dell'interruzione è concluso.
 7. Ripristina tutti i registri salvati sullo stack.
 8. Esegue un'istruzione di **RETURN FROM INTERRUPT** ripristinando lo stato della CPU precedente l'interruzione.

Interruzioni multiple: priorità



I problemi di Intel Architecture-32

- È irrimediabilmente CISC (con formato variabile): le istruzioni possono essere spezzate in istruzioni RISC ma questo richiede tempo e spazio su chip.
- Indirizzamento orientato a memoria (rallenta l'esecuzione se i programmatore non fanno attenzione all'uso delle istruzioni).
- Ci sono pochi registri e non regolari:
 - I risultati intermedi devono transitare per la memoria e si creano molteplici dipendenze.
 - Le istruzioni devono essere eseguite fuori sequenza ed occorre un hardware complesso per riordinarle.
 - Per svolgere queste operazioni occorre una pipeline con molti stadi e quindi molti passaggi prima di completare l'esecuzione di una istruzione; questo implica che la predizione dei salti deve essere precisa (altrimenti finiscono dentro istruzioni che non servono), ma ciò è davvero difficile.

I problemi di Intel Architecture-32

- Per evitare il problema delle predizioni errate si esegue del codice ancor prima che venga richiesto (**esecuzione speculativa**) purtroppo con cause nefaste quando si accede a riferimenti di memoria che non disponibili.
- Disponendo di 32 bit i programmi sono limitati a 4 GB di memoria.

L'architettura IA-64

- Dopo la IA-32, Intel è ha proposto nuove ISA che rompono con il passato:
 - EMT-64 (ISA Pentium a 32 bit esteso a 64).
 - IA-64.
- La IA-64 è una architettura reale a 64 bit sviluppata in collaborazione con HP.
- L'implementazione è una classe di CPU di fascia alta denominata *Itanium*.
- Innovativa ma di poco successo perché il mercato server non ha scelto di abbandonare IA-32.

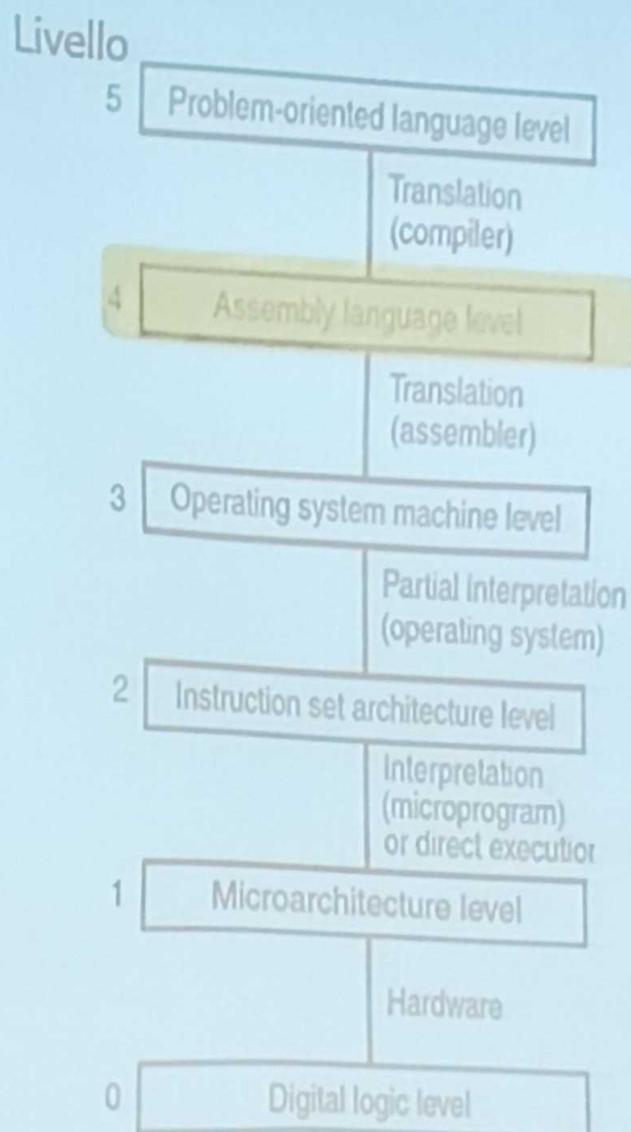
L'architettura IA-64

- Idea di base:
 - spostare il carico di lavoro dall'esecuzione alla compilazione.
- Nel Core i7 durante l'esecuzione il processore:
 - riordina le istruzioni;
 - rinomina i registri;
 - schedula le unità funzionali.
- Il tutto per tenere impegnate tutte le risorse hardware.

L'architettura IA-64

- Nel modello IA-64 invece è il compilatore che prevede tutto ciò senza che ci sia complessità in fase di esecuzione.
- IA-64 ha numerose funzionalità per incrementare le performance:
 1. riduzione degli accessi in memoria,
 2. scheduling delle istruzioni,
 3. riduzione dei salti condizionati,
 4. carichiamenti speculativi.

Il livello del linguaggio assemblativo



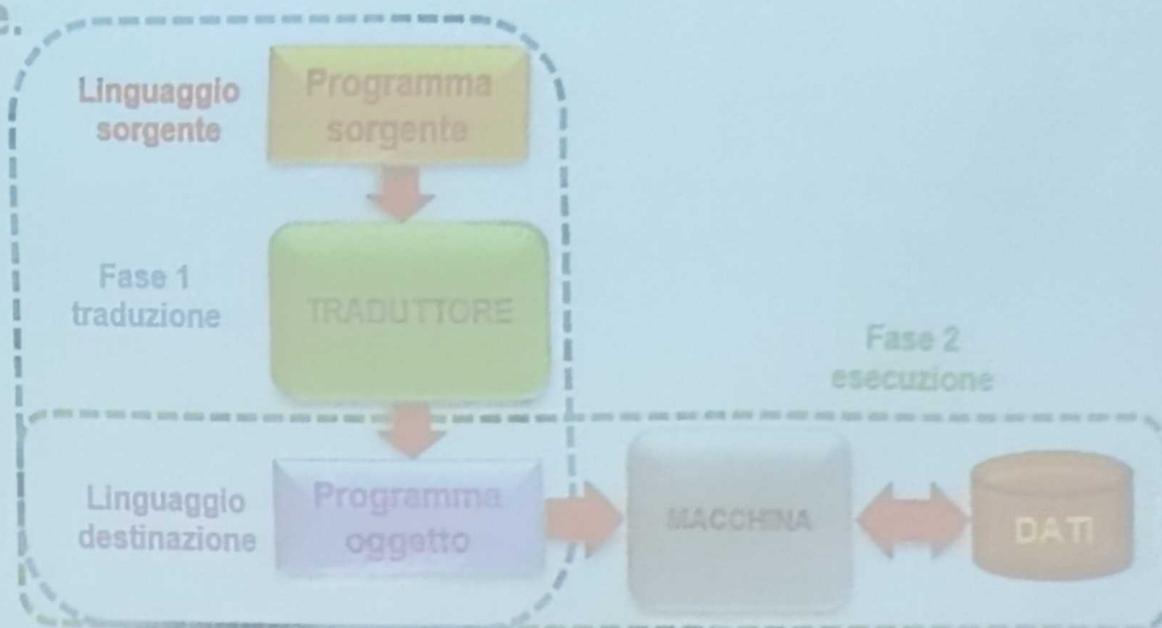
Ricordiamo che:

Assembly è sinonimo di assemblativo e fa riferimento al linguaggio.

Assembler è sinonimo di assemblatore e fa riferimento al traduttore.

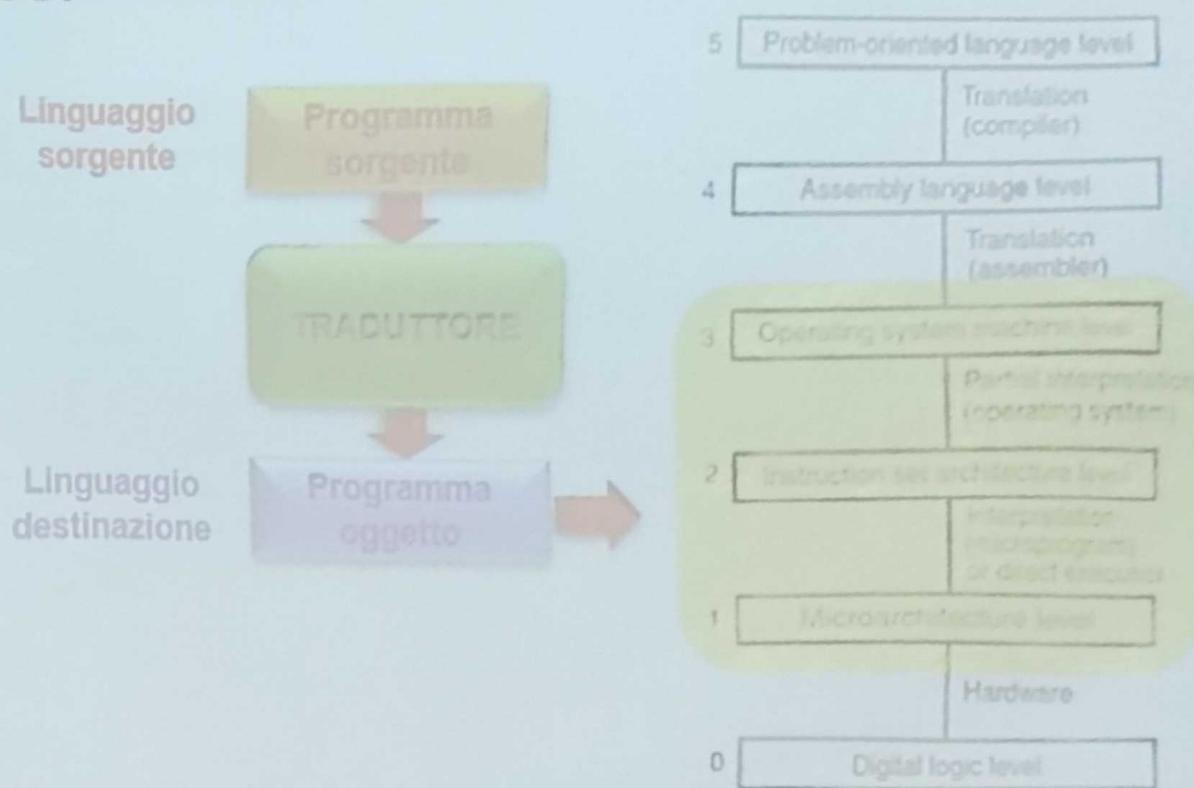
Introduzione

- Mentre il livello di micro e macro-architettura, del sistema operativo sono **interpretati**, il livello del linguaggio assembly è realizzato mediante **traduzione**.
- Il programma che converte un programma sorgente nel corrispondente programma destinazione è detto **traduttore**.



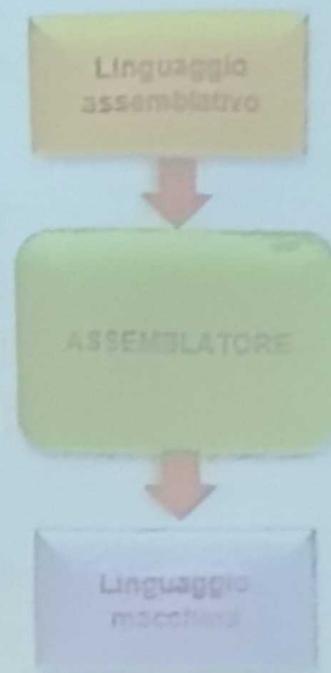
Introduzione

- Mentre si esegue il programma oggetto sono coinvolti i livelli di micro e macro architettura ed il livello macchina del SO.



Introduzione al linguaggio assemblativo

- Il linguaggio sorgente è una rappresentazione simbolica del linguaggio macchina (linguaggio assemblativo)
- Simboli mnemonici (ADD, SUB, MUL,...) sono più semplici da ricordare rispetto ad una sequenza di bit ...010100...
- Il linguaggio assemblativo ha una corrispondenza **uno ad uno** rispetto a quello macchina.
- Il programmatore assembler ha accesso a tutte le risorse della macchina, differentemente da uno che lavora con linguaggi ad alto livello (es. non può vedere i registri).
- Un programma è scritto per una specifica famiglia di macchine, programma ad alto livello è "*machine independent*".



Perché usare il linguaggio assemblativo?

- Programmare in Assembler è complesso.
- Per alcune categorie di applicazioni ove servono ridotte dimensioni del programma, velocità e completo sfruttamento della macchina non esiste alternativa!
- $N = I + J$ eseguito su architettura x86:

Etichetta	Opcode	Operandi	Commenti
FORMULA:			
I	MOV	EAX,I	; EAX = I
J	ADD	EAX,J	; EAX = I + J
	MOV	N,EAX	; N = I + J
	DD	3	; I=3 (I occupa 4 byte)
	DD	4	; J=4 (J occupa 4 byte)
N	DD	0	; N=0 (N occupa 4 byte)

► Define Double (nell'8088 la parola era a 16 bit!)

Pseudoistruzioni

- I comandi che l'assemblatore utilizza nel codice sono dette **pseudoistruzioni o direttive dell'assemblatore**.

Esempi

- Per definire un nuovo simbolo pari al valore di una espressione:

```
BASE EQU 1000
```

- Per allocare 3 byte con dei valori fissati:

```
TABLE DB 11,23,49
```

- Assemblaggio condizionale:

```
WORDSIZE EQU 32
IF WORDSIZE GT 32
    WSIZE DD 64
ELSE
    WSIZE DD 32
ENDIF
```

LE MACROISTRUZIONI

- La definizione di una macro è un modo per assegnare ad un nome una porzione di testo in modo che l'assembler esegua poi la macrosostituzione.

Esempio

- Si vuole scambiare il contenuto di P con quello di Q.

Senza
macro

```
MOV EAX,P  
MOV EBX,Q  
MOV Q,EAX  
MOV P,EBX
```

```
MOV EAX,P  
MOV EBX,Q  
MOV Q,EAX  
MOV P,EBX
```

Con
macro

```
SWAP MACRO  
MOV EAX,P  
MOV EBX,Q  
MOV Q,EAX  
MOV P,EBX  
ENDM
```

```
SWAP  
SWAP
```

Confronto tra macro e procedure

Le macro non devono essere confuse con le procedure!

Questione	chiamata alla Macro	chiamata alla Procedura
Quando è fatta la chiamata?	Durante l'assemblaggio (compile-time)	durante l'esecuzione (run-time)
Il corpo è inserito nel programma oggetto ogni volta che appare la chiamata?	Sì	No
L'istruzione per la chiamata di procedura è inserita nel programma oggetto e successivamente eseguita?	No	Sì
Occorre utilizzare una istruzione di ritorno dopo aver eseguito una chiamata?	No	Sì
Quante copie del corpo appaiono nel programma oggetto?	Tante quante sono le chiamate	Una

Macro con parametri

- La definizione di una macro può includere dei parametri formali che verranno poi sostituiti dai corrispondenti valori attuali.

Esempio

Senza
macro

```
MOV EAX,P  
MOV EBX,Q  
MOV Q,EAX  
MOV P,EBX
```

```
MOV EAX,R  
MOV EBX,S  
MOV S,EAX  
MOV R,EBX
```

```
CHANGE MACRO P1,P2  
MOV EAX,P1  
MOV EBX,P2  
MOV P2,EAX  
MOV P1,EBX
```

```
ENDM
```

```
CHANGE P,Q  
CHANGE R,S
```

Parametri
formali

Macro con
parametri

Parametri
attuali

PROCESSO DI ASSEMBLAGGIO

- Poiché un programma si compone di istruzioni che possono avere dei "salti" in avanti l'assemblatore non può conoscere in anticipo la posizione dell'istruzione richiamata (**problema dei riferimenti in avanti**).
- Esistono due soluzioni:
 - Leggere il programma sorgente due volte costruendo la prima volta una tabella dei simboli, etichette ed istruzioni.
 - Leggere il programma sorgente una volta e convertirlo in un formato intermedio in una tabella in memoria eseguendo poi un secondo passaggio sulla tabella.
- In entrambe le soluzioni il primo passaggio ha il compito di espandere tutte le macro.

Primo passaggio

- Il primo passaggio costruisce la tabella dei simboli.
- Durante questa fase l'assembler utilizza una variabile **ILC** (*Instruction Location Counter*) che memorizza l'indirizzo dell'istruzione che sta assemblando.
- La maggior parte degli assembler utilizza tre tabelle interne per memorizzare:
 - I simboli.
 - Le pseudoistruzioni.
 - I codici operativi.

Esempio

Etichetta	Opcode	Operandi	Commenti	Lung.	+	ILC
MARIA	MOV	EAX,I	;EAX = I	5	100	
	MOV	EBX,J	;EBX = J	6	105	
ROBERTA:	MOV	ECX,K	;ECX = K	6	111	
	IMUL	EAX, EAX	;EAX = I*I	2	117	
	IMUL	EBX, EBX	;EBX = J*J	3	119	
	IMUL	ECX, ECX	;ECX = K*K	3	122	
MARYLYN:	ADD	EAX, EBX	;EAX = I*I+J*J	2	125	
	ADD	EAX, ECX	;EAX = I*I+J*J+K*K	2	127	
STEPHANY:	JMP	DONE	;branch to DONE	5	129	

Tabella dei simboli

Symbol	Value
MARIA	100
ROBERTA	111
MARYLYN	125
STEPHANY	129

Secondo passaggio

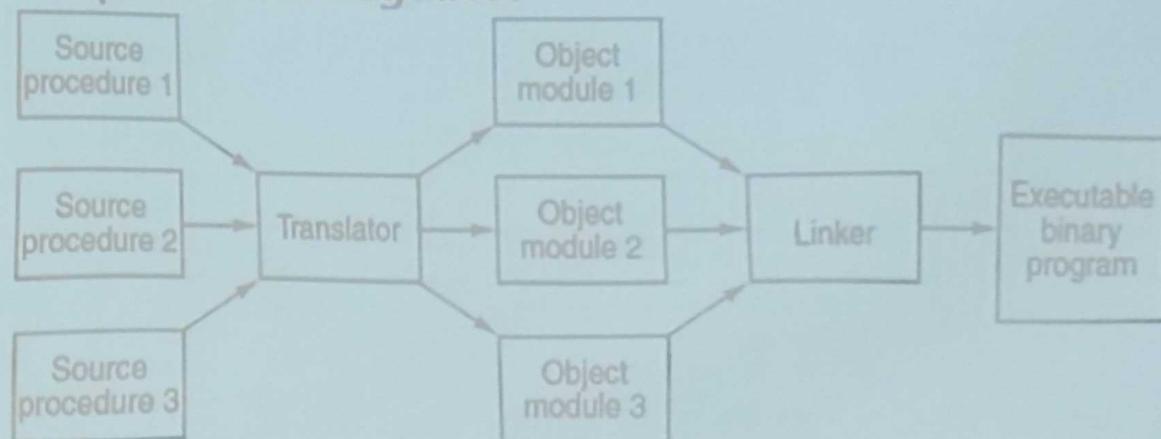
- Durante il secondo passaggio è generato il codice oggetto.
- Deve generare le informazioni utili al **linker** per collegare in un unico file eseguibile tutte le procedure assemblate in momenti distinti.

La tabella dei simboli

- La tabella dei simboli è una tabella associativa: un insieme di coppie <simbolo, valore> accessibili tramite il simbolo.
- Esistono varie tecniche per realizzarla:
 1. Utilizzare una struttura ordinata ed accedervi in modo dicotomico ($O(\log n)$).
 - Mantenere ordinata una struttura di dati costa dal punto di vista computazionale.
 2. Usare una codifica **hash** che mappa i simboli nell'intervallo da 0 a $k-1$ ($O(1)$).
 - Problema dell'uniforme distribuzione della funzione e delle collisioni.

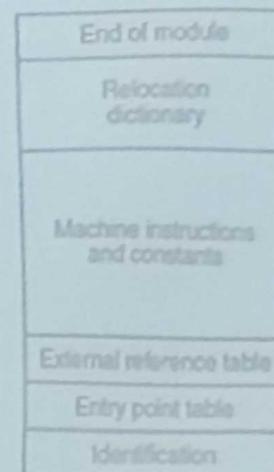
LINKER e LOADER

- Tutti i programmi hanno più procedure.
- Gli assembler traducono una procedura alla volta in linguaggio oggetto e salvano il risultato sul disco.
- Prima di poter eseguire il programma occorre collegare in modo appropriato tutte le procedure oggetto (**linking**).
- In assenza di memoria virtuale occorre caricare, attraverso il **loader**, il programma generato in memoria centrale prima di eseguirlo.



La struttura di un codice oggetto

- Inizialmente troviamo il codice identificativo e le lunghezze delle singole parti del modulo (dati utili al linker).
- Segue l'insieme dei punti di ingresso a cui possono fare riferimento altri moduli (direttiva PUBLIC).
- Troviamo poi la lista dei riferimenti utilizzati all'esterno del modulo (es. richiami a procedure esterne attraverso la direttiva EXTERN).
- A questo punto troviamo il codice assemblato e le costanti (l'unica parte che verrà caricata in memoria al momento dell'esecuzione).
- Segue il dizionario di rilocazione che fornisce gli indirizzi che dovranno essere rilocati.
- Infine troviamo l'identificativo di fine modulo, l'indirizzo ove iniziare l'esecuzione, un eventuale *checksum* per rilevare gli errori che possono avvenire durante la lettura del modulo



Conclusioni

- Affrontate le problematiche inerenti le modalità di indirizzamento nel livello ISA.
- Individuato le tipologie di istruzioni del livello ISA.
- Introdotto le caratteristiche delle architetture IA-32 e IA-64.
- Analizzato il livello del linguaggio assemblativo.
- Le modalità di assemblaggio e di esecuzione.