Programmazione Assembler

Giovanni Stea

Reti Logiche

Corso di Laurea in Ingegneria Informatica, Universita' di Pisa

a.a. 2020/21

Fare clic per inserire le note

Assembler o Assembly

- Assembly piu' corretto
- Ma noi diremo Assembler
  - Ragioni storiche

- Il linguaggio con cui si scrivo ttamente **le istruzioni eseguite dal processore**
- Il processore, ciclicamente:
  - preleva un'istruzione macchi memoria
  - La esegue

Linguaggio macchina e Assembler

alto livello

a = b ; basso liv.

- Cosa contiene la memoria?
  - Zeri e uni
- Il **linguaggio macchina** e' una sequenza di zeri ed uni
  - Incomprensibile ad un utente umano
- Assembler: sintassi **simbolica** (human-friendly) per il linguaggio macchina
  - **Assembler** **linguaggio macchina**
  - MOV %AX, %BX -> numero binario 0100
- Il passaggio dall'Assembler al LM si chiama **assemblaggio**
  - Cosa diversa dalla **compilazione** (e.g., del C++). Traduzione 1:1

Generalita' sull'assembler

- Mancano costrutti di controllo di flusso strutturato tipici dei linguaggi ad alto livello
  - Niente `for, while, do…while, if…then…else, case`
  - Solo costrutti primitivi (salti)
  - Maggior sforzo da parte del programmatore (pensare come la macchina, «**a basso livello**»)

10110011

JMP

- Mancano tipi di variabile
  - Gli operandi sono stringhe di bit
  - L'interpretazione delle stringhe di bit e' nella testa del programmatore, non nei costrutti del linguaggio

Generalita' sull'assembler

- L'Assembler e' **processor-specific**
  - Cosi' come lo sono I set di istruzioni dei processori
  - Assembler per Intel x86, Arm Cortex, etc.
- Codice Assembler mai portabile
- Principi generali validi
- Utilizzo nei sistemi embedded
- Validità culturale

- Assembler Intel x86
  - Non tutto (manuale ~2000 pagine)
  - Alcune istruzioni, quanto basta per poter scrivere programmi sensati

Schema a blocchi del calcolatore

- Modello funzionale

Arithmetic Logic Unit
$\mathbb{N}$ $\mathbb{Z}$

Immagini, suoni, etc.

Rappresentazione di numeri naturali e interi

- ALU esegue
  - operazioni **logiche** (AND, OR, NOT, etc.) su stringhe di bit
  - operazioni **aritmetiche**, interpretando le stringhe di bit che maneggia come
    - numeri naturali in base 2
    - numeri interi rappresentati in complemento a 2

- Rappresentazione dell'informazione
  - Argomento gia' affrontato in altri esami
  - Breve recap

Numeri naturali

110101

32+16+4+1=53

- Su *N* bit si rappresentano 2^N numeri naturali $\left[0; 2^N - 1\right]$
- Stringa di bit <-> numero in base due

$$b_{N-1}, b_{N-2}, \ldots, b_1, b_0 \Leftrightarrow X = \sum_{i=0}^{N-1} b_i \cdot 2^i$$

Most Significant Bit (MSB)

Least Significant Bit (LSB)

53	1
26	0
13	1
6	0
3	1
1	1
0	

- Algoritmo delle divisioni successive per trovare le cifre in base due

Numeri interi in complemento a due

- Su *N* bit si rappresentano 2^N numeri ~~naturali~~ interi $\left[-2^{N-1}; 2^{N-1}-1\right]$
- Su 8 e 16 bit l'intervallo è $[-128, 127]$, $[-32768, 32767]$
- Il numero **intero** x viene **rappresentato con la stringa di bit X** che corrisponde al numero naturale

$$X = \begin{cases} x & x \geq 0 \\ 2^N + x & x < 0 \end{cases}$$

- Purche' x sia un intero rappresentabile, i

$$x \in \left[-2^{N-1}; 2^{N-1}-1\right]$$

- Versione equivalente: $X = |x|_{2^N}$

Numeri interi in complemento a due

$$X = \begin{cases} x & x \geq 0 \\ 2^N + x & x < 0 \end{cases}$$

$$x \in \left[-2^{N-1}; 2^{N-1} - 1\right]$$

Quando si parla di numeri in C2, tenere **sempre** a mente questo disegno

Numeri interi in complemento a due

- Legge inversa (trovare il numero intero x dalla sua rappresentazione X)

$$x = \begin{cases} X & X_{N-1} = 0 \\ -\left(\bar{X} + 1\right) & X_{N-1} = 1 \end{cases}$$

- Il *complemento* è l'operazione che cambia gli "0" in "1" e viceversa

C2
N = 8

1011 0101
$-(0100\,1010 + 1)$
$-(0100\,1011)$
$-(64 + 8 + 2 + 1) = -75$

Notazione esadecimale

11

1011 1001

B 9

- Sequenze lunghe di zeri e uni sono difficili da comprendere
- 4 bit sono un numero tra 0 e 15. Possiamo associare un simbolo **esadecimale** a una stringa di 4 bit

Hexadecimal

0x 37

Quadr.	Simb.	Quadr.	Simb.	Quadr.	Simb.	Quadr.	Simb.
0000	0	0100	4	1000	8	1100	C
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	F

- Si scriverà quindi `0xC1` per indicare il byte `1100 0001`
- `0x` prefisso che indica che un numero esadecimale

Esercizi (per casa)

- rappresentare i seguenti numeri naturali su 8 bit: 32, 63, 130
- per ciascun numero x rappresentato, chiedersi quale sia la rappresentazione di $2x$, $4x$, … $2^k x$, e quale quella del quoziente della divisione intera $x/2$, $x/4$, …$x/2^k$. Ricavare una regola generale valida per i numeri naturali.

- rappresentare i numeri interi (su 8 bit in complemento a 2) -32, -1, -128, 127
- per ciascun numero x rappresentato, chiedersi quale sia la rappresentazione di $2x$, $4x$, … $2^k x$, e quale quella del quoziente della divisione intera $x/2$, $x/4$, …$x/2^k$. Ricavare una regola generale valida per i numeri interi.

- Rappresentare i numeri interi di cui alla precedente domanda su <u>16</u> bit in complemento a 2.
- Qual è la relazione tra la loro rappresentazione su 8 bit e la loro rappresentazione su 16 bit? Ricavare una regola generale per ottenere la seconda dalla prima.

Struttura del calcolatore

- Visto da un programmatore Assembler

- **Spazio di memoria**
  - Spazio lineare di 2^{32} **locazioni** (**celle**) contigue, di capacita' **un byte ciascuna**
  - Ogni cella identificata da un numero a 32 bit, detto **indirizzo**
  - Il processore puo' accedere (leggere/scrivere) a
    - Singola locazione (operando a 8 bit)
    - Doppia locazione (16 bit)
    - Quadrupla locazione (32 bit)

istruzioni
operandi

RAM volatile
ROM

Spazio di Memoria

- Acessi a 16/32 bit: si usa l'indirizzo piu' piccolo delle 2/4 locazioni
- Indirizzo piu' grande contiene bit piu' significativi

Lettura a	Indirizzo	Contenuto
8 bit (byte)	0x3F65B432	0x1C
16 bit (word)	0x3F65B432	0x391C
32 bit (double word)	0x3F65B432	0xC6A2391C

2^{10} K 2^{20} M 2^{32} $2^{30} \cdot 2^{2}$

Spazio di memoria

2^{30} G 4Gb

- Implementato in tecnologia RAM (volatile)
  - Contenuto casuale al reset
- Tranne una piccola parte, che **deve** essere ROM (permanente)
  - Altrimenti il processore non ha istruzioni da prelevare al reset

pippo

- E se nel sistema non ci sono 4Gb (=2^{32})?
- Come faccio a sapere se posso riferire un indirizzo?
  - Potrebbero esserci dati non miei (e.g., sistema operativo)
- In Assembler gli indirizzi sono **simbolici**
  - La traduzione viene fatta in parte dall'assemblatore, in parte a run time
  - Non e' un problema per il programmatore

Spazio di I/O

2^{10} 2^{6}

- $2^{16} = \mathbf{64}\boldsymbol{k}$ **locazioni o *porte***.
- Ogni porta ha una capacità pari ad **un byte** ed è indirizzabile attraverso un ***indirizzo* a 16 bit**.

- Processore: accede in lettura o scrittura, usando due particolari istruzioni

  IN OUT

  - Spesso una porta **solo** lettura o **solo** scrittura

- Le locazioni di memoria sono identiche, le porte di I/O no.
  - All'indirizzo della tastiera c'e' la tastiera, non il mouse
  - E' necessario conoscere gli indirizzi

Processore

E extended

16 bit → 32 bit

- Registri: piccole locazioni di memoria interne (32 bit)
  - **Generali**: per le elaborazioni
  - **Di stato**: riservati a compiti speciali

- 16 bit bassi dei registri generali riferibili autonomamente (retro-compatibilita')
- Di alcuni registry si possono riferire parti ad 8 bit

Registri generali

- Alcuni registri utilizzati per particolari funzioni (motivi storici)

- EAX (AX, AH od AL) è utilizzato da alcune istruzioni aritmetiche per contenere operandi e risultati. A volte viene chiamato **accumulatore**.
- ESI, EDI, EBX ed EBP sono a volte utilizzati come **registri puntatore**, base (B) e indice (I)
- ESP è utilizzato per indirizzare la **pila** o **stack**. stack pointer
  - parte di memoria con disciplina LIFO
  - Serve per gestire **sottoprogrammi**

Registri generali

- **EAX: Accumulator.** È il registro che serve per fare calcoli aritmetici. Alcune istruzioni aritmetiche lo usano per contenere operandi e risultati
- **EBX: Base.** Veniva spesso usato come indirizzo di base per l'accesso in memoria
- **ECX: Counter.** Viene usato come contatore nei cicli (la variabile "i" del for viene spesso mappata sul regist parti di esso)
- **EDX: Data.** Anche questo ato come operando da istruzioni aritmetiche.
- **ESI: Source Index.** Veniva me registro indice per accessi in memoria
- **EDI: Destination Index.** C ra
- **EBP: Base pointer.** Veniva me registro "base" per accessi in memoria.

- E = extended (prima erano registri a 16 bit, poi si e' passati a 32 bit)

Registri di stato

- EIP (*Instruction Pointer register*, a volte anche chiamato **program counter**)
  - l'indirizzo della locazione a partire dalla quale sarà prelevata la **prossima** istruzione da eseguire
  - Contenuto fissato al reset iniziale
    - E' nota la prima istruzio ocessore esegue (deve stare in ROM)
- Quindi, il processore
  - **preleva** dalla memoria, zo EIP, una nuova istruzione.
  - **incrementa** EIP del num te dell'istruzione che ha prelevato
  - **esegue** quell'istruzione, al passo precedente.
- Le istruzioni vengono e **una dopo l'altra** nell'ordine in cui sono scritte in memoria
  - Tranne quando si incontrano «salti», istruzioni che modificano EIP

Registri di stato

- EF (*Extended Flag register*) ha 32 elementi detti *flag*.
  - **OF (*Overflow Flag*)**: se 1, durante l'esecuzione dell'ultima istruzione si è avuto un ***traboccamento***: se l'istruzione operava su numeri interi, il risultato dell'operazione aritmetica non è rappre[illegible].
  - **SF (*Sign Flag*)**: se 1, l'ult[illegible]zione ha generato un risultato con MSB=1: se l'istruzione operava su [illegible]teri, il risultato è un numero negativo.
  - **ZF (*Zero Flag*)**: se 1, l'ul[illegible]zione ha generato un risultato con tutti i bit a 0.
  - **CF (*Carry Flag*)**: se 1 l'ul[illegible]uzione ha generato un ***riporto*** o un ***prestito***: se l'istruzione operava su [illegible]aturali, il risultato dell'operazione aritmetica non è rappresentabile.

- Operazioni su numeri **interi**: guardare OF, SF, ZF (~~CF~~ non significativo)
- Operazioni su numeri **naturali**: guardare CF, ZF (~~OF~~, ~~SF~~ non significativi)

Condizioni al reset

- EIP vale 0xFFFF0000
  - La prima istruzione si trova li’
  - Le celle da 0xFFFF0000 in poi devono essere implementate in ROM

- In EF I flag che ci interessano valgono 0.

Codifica macchina e codifica mnemonica

- codifica **macchina**: serie di zeri e di uni che codificano le istruzioni che il processore esegue
- codifica **mnemonica**: modo **simbolico** di scrivere quelle serie di zeri e di uni.
- linguaggio **Assembler** usa codifica mnemonica delle istruzioni, e dispone di **una serie di sovrastrutture sintattiche** che lo rendono più semplice da usare per un programmatore.
  - Per esempio **nomi simbolici** per le locazioni di memoria

Codifica macchina e codifica mnemonica

- Il formato macchina e' il seguente:

I prefix 0/1 byte	O prefix 0/1 byte	Opcode 1/2 byte	Mode 0/1 byte	SIB 0/1 byte	Displacement 0/4 byte	Immediate 0/4 byte

- E' abbastanza irrilevante, tanto noi scriveremo:

GNU Assembler GAS

```
MOV %EAX, 0x01F4E39A
```

- E l'assemblatore traduce nel formato di sopra
  - Secondo regole sue, che non abbiamo bisogno di conoscere
- Meglio ancora, potremo scrivere

```
MOV %EAX, pippo
```

- `pippo` e' un nome simbolico, che riferisce un qualche indirizzo di memoria

ADD_

Istruzioni in codifica mnemonica

MOV_

- Un'istruzione consta di **tre campi**
  - **codice operativo**: stabilisce quale è l'operazione che deve essere eseguita (e.g., somma)
  - **suffisso di lunghezza**: stabilisce quanto sono lunghi gli operandi dell'istruzione (e.g., byte), quando un'istruzione ammette operandi di varia lunghezza.
  - **operandi** dell'istruzione (e.g., gli addendi).

byte 8
Word 16
long 32

```
OPCODEsuffix source, dest
ADDW %BX, pluto
```

- Gli operandi possono essere contenuti in **registri**, celle di **memoria**, porte di **I/O, direttamente nell'istruzione (costanti)**

Istruzioni in codifica mnemonica

ADD ~~$~~ %BX, pluto

pluto += %BX

- Le istruzioni ammettono **zero, uno o due** operandi.
- Se due, il primo e' **sorgente**, il secondo e' **destinatario**
  - Il sorgente non viene modificato
  - Il destinatario viene sostituito con il risultato dell'elaborazione
- Se uno, puo' essere sorgente/destinatario a seconda dell'istruzione
  - E' chiaro dal contesto

~~ADD %BX, %EAX~~

- Se ci sono due operandi **sono della stessa lunghezza** (e.g., entrambi 32 bit)
  - Tranne qualche eccezione che vedremo

Primo esempio di programma

- **preleva** dalla memoria un operando a 32 bit, che si trova a partire dall'indirizzo 0x00000100
- **conta** il numero di bit a 1 nell'operando
- **scrive** il risultato nella locazione di memoria di indirizzo 0x00000104

Indirizzo	Contenuto
0x00000100	
0x00000101	
0x00000102	
0x00000103	
0x00000104	
…	
0x00000200	MOVB $0x00, %CL
0x00000202	MOVL 0x00000100, %EAX
0x00000207	CMPL $0x00000000, %EAX
0x0000020A	JE %EIP+$0x07
0x0000020C	SHRL %EAX
0x0000020E	ADCB $0x00, %CL
0x00000211	JMP %EIP-$0x0C
0x00000213	MOVB %CL, 0x00000104
0x00000218	…

Src Dest

Primo esempio di programma (cont.)

Indirizzo	Contenuto
0x00000100	LSB
0x00000101	
0x00000102	
0x00000103	MSB
0x00000104	

…

segfault

```
0x00000200   MOVB $0x00, %CL
0x00000202   MOVL 0x00000100, %EAX
0x00000207   CMPL $0x00000000, %EAX
0x0000020A   JE %EIP+$0x07
0x0000020C   SHRL %EAX
0x0000020E   ADCB $0x00, %CL
0x00000211   JMP %EIP-$0x0C
0x00000213   MOVB %CL, 0x00000104
0x00000218   …
```

dopo

dopo

scrive EF

JUMP if equal

Shift right

Add with Carry

JUMP

CL += Ø + CF

Istruzioni Assembler

- **operative**: quelle che svolgono qualche compito (ADC, SHR, MOV, CMP)
- **di controllo**: quelle che alterano il flusso (sequenziale) del programma (JMP, JE)

- Tante istruzioni operative (le vedremo)
- Le istruzioni di controllo sono piu' o meno tutte qui
  - Si salta: JMP
  - Si salta o no, sulla base del contenuto di un flag (JE,…)

While

Indirizzamento delle istruzioni operative

`OPCODEsize source, destination`

- `OPCODE`: codice operativo (e.g. MOV, CMP)
- `Size`: spesso puo' essere omesso (se chiaro dagli operandi)
- `Source, destination`: individuati tramite **modalita' di indirizzamento**

SHR %EAX

pippo

`OPCODEsize operando`

`OPCODE`

Indirizzamento di registro

- Uno tra:
  - 8 registri **generali** a 32 bit (EAX, EBX, ECX, EDX, EBP, ESI, EDI e ESP),
  - 8 registri generali a 16 bit (AX, BX, CX, DX, SI, DI, SP, BP)
  - 8 registri generali ad 8 bit (AH, BH, CH, DH, AL, BL, CL, DL)

- Si applica sia al sorgente che al destinatario
- Esempi

```
OPCODE %DI
OPCODE %EAX,%EBX
OPCODE %AH,%CL
```

16bit

32bit

8bit

suffisso

Indirizzamento immediato

- **Solo** per l'operando **sorgente**

```
OPCODE $0x20,%AL
OPCODE $0x5683A20B,%ECX
```

8bit
32bit

Va inserito il $ (non dimenticarlo)

Indirizzamento di memoria

- **O sorgente, o destinatario** (non entrambi)
- Necessario specificare un indirizzo di memoria a 32 bit

FPU

- Caso piu' generale

EBX EDI
EBP ESI

1 (default), 2, 4, ~~8~~

indirizzo = |base + indice × scala ± displacement|$_{\text{modulo}\,2^{32}}$

ESP

Registro generale a 32 bit

Costante intera

- La sintassi è:

Cost rg rg Cost

```
OPCODEsfx ±disp(base,indice,scala)
```

Indirizzamento di memoria

- Di tipo **diretto**

  MOV 0x ... 100, %EAX

  - utilizzato unicamente il displacement, che coincide con l'indirizzo.

```
OPCODEW 0x00002001
```

- Operando a 16 bit (suffisso W), che si trova nella doppia locazione il cui indirizzo piu' basso e' 0x00002001.
- Suffisso necessario, altrimenti non potrei sapere se l'operando e' a 8, 16, 32 bit
- Esadecimali **senza il $** sono interpretati come **indirizzi di memoria**, non costanti
  - Errore tipico, risultato **segmentation fault**

Indirizzamento di memoria

%EBX

- **Indirizzamento indiretto** (o con *registro puntatore*)

base

```
OPCODEL (%EBX)
```

- operando a 32 bit, si trova nella quadrupla locazione il cui indirizzo è contenuto nel registro EBX

indice

```
OPCODEL (,%ESI,4)
```

- operando a 32 bit, si trova nella quadrupla locazione il cui indirizzo è dato dal contenuto del registro indice ESI moltiplicato per 4
  - Attenzione alla scala: moltiplica l'indice (che e' il **secondo registro**), non la base (che e' il primo)

Indirizzamenti di memoria

- **Indirizzamento con displacement e registro di modifica**

base

```
OPCODEW 0x002A3A2B(%EDI)
```

- operando a 16 bit, si trova nella doppia locazione il cui indirizzo si ottiene sommando, modulo 2^{32},
  - il displacement 0x002A3A2B
  - il contenuto di EDI
- **Indirizzamento bimodificato senza displacement**

```
OPCODEW (%EBX, %EDI)
OPCODEW (%EBX, %EDI, 8)
```

Indirizzamenti di memoria

- **Indirizzamento bimodificato con displacement**

```
OPCODEB 0x002F9000(%EBX,%EDI)
```

- l'operando è ad 8 bit e si trova nella locazione il cui indirizzo vale:

base in EBX + indice in EDI + 0x002F9000|modulo2^{32}

```
OPCODEB -0x9000(%EBX,%EDI)
```

- l'operando è ad 8 bit e si trova nella locazione il cui indirizzo vale:

|base in EBX + indice in EDI – 0x00009000|modulo2^{32} =

Indirizzamento delle porte di I/O

- Uno dei due operandi (**mai entrambi**) può trovarsi nello spazio di I/O
- Indirizzamento **diretto**
  - Si puo' usare solo per indirizzi <256 8bit
  - Perche' nel formato macchina ci sono 8 bit
- Indirizzamento **indiretto con registro puntatore**
  - Si puo' usare solo il registro DX

→ IN
→ OUT 16 bit

```
IN 0x001A,%AL        8
IN (%DX),%AX         16bit
OUT %AL,0x003A       8
OUT %AL,(%DX)        8
```

Note sulla sintassi

GAS MASM

- gli operandi immediati (le costanti) **vanno preceduti dal dollaro $**
- i registri **vanno preceduti dal simbolo di percentuale %**
- in un'istruzione, un numero non preceduto dal dollaro è **un indirizzo di memoria (indirizzamento diretto)**, non una costante.
- Ad esempio:

```
MOV 0x00002000,%EDX
MOV $0x00002000,%EDX
```

- Entrambe lecite, fanno cose diverse

Note sulla sintassi

MOVB $0x2050, %AX

MOV $0x50, %AL

- Suffissi: B (byte, 8 bit), W (word, 16), L (long,32)
- Possono essere omessi se uno degli operandi e' **un registro**
  - In questo caso, la dimensione del registro rende l'istruzione non ambigua

- Altrimenti va specificato (l'assemblatore *dovrebbe* segnalare errore)

```
MOVW $0x2000, (%EDI)
MOVW $0x10, 0x2000
```

0010 00002000

- Non fate troppo conto sull'aiuto dell'assemblatore
  - Spesso l'unica possibilita' e' il debugging

op indirizzo diretto di registro

Principali istruzioni

- istruzioni **operative:**
  - istruzioni di trasferimento
  - istruzioni aritmetiche
  - istruzioni di traslazione/rotazione
  - istruzioni logiche
- istruzioni **di controllo**.
  - Istruzioni di salto
  - Istruzioni per la gestione di sottoprogrammi

- Fa comodo avere la dispensa sottomano

Descriviamo:

- il formato
- cosa fa
- se aggiorna i flag, quali e come
- quali sono le modalità di indirizzamento ammesse per gli operandi

Istruzioni di trasferimento

- “spostano” bit da una parte a un’altra
  - Memoria <-> registro
  - Registro -> registro
  - I/O <-> registro

- Non esistono altre possibilita’
  - e.g., non esiste trasferimento memoria -> memoria.
  - Si passa da un registro del processore, ci vogliono due istruzioni
  - Salvo eccezioni che vedremo piu’ avanti (istruzioni stringa)

- Nessuna di queste modifica I flag

MOVE

- **FORMATO**: MOV source, destination
- **AZIONE**: Sostituisce l'operando destinatario con una copia dell'operando sorgente
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Operandi	Esempi
Memoria, Registro Generale	MOV 0x00002000,%EDX
Registro Generale, Memoria	MOV %CL,0x12AB1024
Registro Generale, Registro Generale	MOV %AX,%DX
Immediato, Memoria	MOVB $0x5B,(%EDI)
Immediato, Registro Generale	MOV $0x54A3,%AX

MEM, REG

LOAD EFFECTIVE ADDRESS

~~LEA~~ S

LEA expr, %EAX

MOV $expr, %EAX

- **FORMATO**: LEA source, destination
- **AZIONE**: Sostituisce l'operando destinatario con l'espressione indirizzo contenuta nell'operando sorgente.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Operandi	Esempi
Memoria, Registro Generale a 32 bit	LEA 0x00002000,%EDX LEA 0x00213AB1(%EAX,%EBX,4),%ECX

1) → MOV 0x00213AB1(%EAX,%EBX,4),%ECX

2) LEA 0x00213AB1(%EAX,%EBX,4),%ECX

EXCHANGE

- **FORMATO**: `XCHG source, destination`
- **AZIONE**: Sostituisce all'operando destinatario una copia dell'operando sorgente e viceversa.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Operandi	**Esempi**
Memoria, Registro Generale	XCHG 0x00002000,%DX
Registro Generale, Memoria	XCHG %AL,0x000A2003
Registro Generale, Registro Generale	XCHG %EAX,%EDX

- La XCHG **modifica il sorgente**. È l'unica istruzione che lo fa.
- Fatto curioso (per adesso): in Assembler si possano scambiare due operandi con una sola istruzione (**non trasparenza** dei registri)

INPUT

16bit

- **FORMATO**: <256

```
IN indirizzo, %AL
IN indirizzo, %AX
IN (%DX),%AL
IN (%DX),%AX
```

8bit

16 bit

- **AZIONE**: Sostituisce il contenuto del registro destinatario (AL, AX) con il contenuto di un adeguato numero di porte consecutive. L'indirizzo della prima (ed eventualmente unica) porta è specificato direttamente nell'istruzione (primi due formati) o è contenuto nel registro DX (ultimi due formati); i primi **due** formati possono essere utilizzati solo per individuare porte con indirizzo inferiore a 256.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

OUTPUT

- **FORMATO**:

  <256

  ```
  OUT     %AL, indirizzo
  OUT     %AX, indirizzo
  OUT     %AL,(%DX)
  OUT     %AX,(%DX)
  ```

  8

  16

- **AZIONE**: Copia il contenuto del registro sorgente (AL, AX) in un adeguato numero di porte consecutive. L'indirizzo della prima (ed eventualmente unica porta) è specificato direttamente nell'istruzione (primi due formati) o è contenuto nel registro DX (ultimi due formati); i primi tre formati possono essere utilizzati solo per individuare porte con indirizzo inferiore a 256.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

INPUT, OUTPUT, non ortogonalità

- Uniche due istruzioni che riguardano l'I/O
  - Gli operandi nell'I/O si possono solo **trasferire** nei/dai registri generali
  - Eventuali operazioni si fanno nel processore, non nelle porte
- Input ed output usano **soltanto**
  - i registri AL, AX, ~~EAX~~ come sorgente/destinatario
  - DX come registro puntatore
- L'Assembler **non è ortogonale**
  - Il C++ è ortogonale. Se la sintassi mi consente di scrivere una variabile, ci posso mettere qualunque variabile
  - L'Assembler no. Se posso fare una cosa con un registro generale, non è detto che la possa fare con gli altri

BX → 0x3142

```
Mov %Bx, %Ax
Mov $0x3142, %Dx
Out %Ax, (%Dx)
```

La pila (stack)

- Zona di memoria gestita **Last In First Out (LIFO)**, vista in esami precedenti
- **Essenziale** al funzionamento del calcolatore
  - Serve a poter annidare sottoprogrammi
  - L'Assembler è organizzato per sottoprogrammi
  - Prima di "saltare" al sottoprogramma **salvo l'indirizzo di ritorno** (i.e. quello della prossima istruzione) **nella pila (PUSH)**
  - Al termine del sottoprogramma, per tornare dove ero, faccio una **POP** dalla pila

La pila

stack Pointer

- Il registro ESP serve come **puntatore al top della pila**
  - Quindi **non va usato** per altri scopi
- PUSH valore
  - Decrementa ESP
  - Copia valore in (%ESP)
- POP destinatario
  - Copia (%ESP) in destinatario
  - Incrementa ESP
- ESP va inizializzato prima che parta il programma

PUSH

- **FORMATO**: `PUSH source`
- **AZIONE**: Salva nella pila corrente una copia dell'operando sorgente (che deve essere a **16 o a 32 bit**). Più in dettaglio, compie le seguenti azioni: i) decrementa l'indirizzo contenuto nel registro ESP di due o di quattro; ii) memorizza una copia dell'operando sorgente nella doppia o quadrupla locazione il cui indirizzo è contenuto in ESP.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Operandi	**Esempi**
Memoria	PUSHW 0x3214200A
Immediato	PUSHL $0x4871A000
Registro Generale	PUSH %BX

POP

POP %AL

- **FORMATO**: `POP destination`
- **AZIONE**: Rimuove dalla pila corrente una word o un long e la sostituisce all'operando destinatario. Più in dettaglio, compie le seguenti azioni: i) sostituisce all'operando destinatario una copia del contenuto della doppia o della quadrupla locazione il cui indirizzo è contenuto in ESP, ii) incrementa di due o di quattro l'indirizzo contenuto in ESP, rimuovendo in tal modo dalla pila la word o il long copiato.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Operandi	**Esempi**
Memoria	POPW 0x02AB2000
Registro Generale	POP %BX

La pila come memoria temporanea

- I registri generali sono pochi
- Alcuni registri generali (e.g. EAX) sono gli unici operandi possibili in alcune istruzioni
- Succede spesso che ho devo usare un registro per due scopi diversi in momenti diversi
- **«Parcheggio»** i dati in pila e li riprendo

```
#sto usando %EAX, mi serve un dato da una porta
PUSH %EAX
IN 0x001A, %AL
…
POP %EAX            # riprendo I conti che stavo facendo
```

All double

PUSHAD/POPAD

- **FORMATO**: `PUSHAD`
- **AZIONE**: Salva nella pila corrente una copia del contenuto degli 8 registri generali a 32 bit, rispettando il seguente ordine: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

- **FORMATO**: `POPAD`
- **AZIONE**: Rimuove dalla pila 8 long e con essi rinnova il contenuto degli 8 registri generali a 32 bit, rispettando il seguente ordine: EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX.
- **FLAG** di cui viene modificato il contenuto: Nessuno
  - Si noti che ESP **non viene sovrascritto** (sarebbe un problema se lo fosse).

Istruzioni Aritmetiche

- **Alcune** (**non tutte**) considerano gli operandi indifferentemente come **naturali** e come **numeri interi**,
  - L'istruzione di **somma** esegue quindi l'algoritmo giusto quale che sia l'interpretazione delle stringhe di bit che le vengono passate come operandi.
- **Modificano i flag**
  - Considerano due algoritmi: uno che va bene per operandi **naturali**, ed uno che va bene per **interi**
  - I flag da guardare sono diversi (CF per naturali, SF/OF per interi)
- Il programmatore è **l'unico che sa cosa sta scrivendo**
  - Se una certa stringa di bit è un naturale o un intero
  - il programmatore farà seguire queste istruzioni da qualche altra istruzione che va a guardare i **flag giusti**.

ADD

dest += source

- **FORMATO**: `ADD source, destination`
- **AZIONE**: Modifica l'operando destinatario sommandovi l'operando sorgente (il risultato dell'operazione è consistente sia se gli operandi sono interpretati come numeri naturali sia se gli operandi sono interpretati come numeri interi); mette ad 1 il contenuto del flag CF se, interpretando gli operandi come numeri naturali, si è verificato un riporto; mette ad 1 il contenuto del flag OF se, interpretando gli operandi come numeri interi, si è verificato un traboccamento.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	**Esempi**
Memoria, Registro Generale	ADD 0x00002000,%EDX
Registro Generale, Memoria	ADD %CL,0x12AB1024
Registro Generale, Registro Generale	ADD %AX,%DX
Immediato, Memoria	ADDB $0x5B,(%EDI)
Immediato, Registro Generale	ADD $0x54A3,%AX

ADD (cont.)

- è come se scrivessi `dest += src` in C++
- Le somme si fanno nello stesso modo in qualunque base:
- Supponiamo operandi a 3 cifre per semplicità

Naturali

Riporto (Carry) uscente.
Finisce in CF

ADD (cont.)

interi

- Il **risultato** è corretto sia per **naturali** che per **interi, <u>I flag no</u>**
- Se fossero interi, il primo sarebbe la rappresentazione di

$$-\left(\overline{101}+1\right)=-\left(010+1\right)=-011$$

-3
+3

- E quindi, la somma sarebbe (-3) +3 = 0
- Il risultato è corretto e rappresentabile
- Il fatto che CF=1 è irrilevante (sono interi)
  - Se somma di interi, CF **non va guardato**

ADD (cont.)

OF

- Come si fa a sapere se la somma di due **interi** è corretta?
  - Si ha **overflow** quando il risultato esce dall'intervallo di rappresentabilità
- Si guardano i **segni di operandi e risultato**

OF=Ø

  - Operandi **discordi:** risultato **sempre corretto**
  - Operandi **concordi:** risultato corretto se concorde con gli operandi
- La ADD mette OF a 1 quando
  a) gli operandi sono concordi; e
  b) il risultato è discorde con gli operandi

ADD (cont.)

interi

$-(010+1) = -011 = -3$

- Sempre numeri rappresentati su 3 bit:

[0:7]

+3
+3
+6

CF

0	1	1	
	0	1	1 +
	0	1	1
	01	11	10
	1	**1**	**0**

$[-2^2 ; +2^2 - 1]$
$[-4 ; +3]$

OF = 1

naturali

5+
5
10

CF

1	0	1	
	1	0	1 +
	1	0	1
	10	01	10
	0	**1**	**0**

OF = 1

-3+
-3
-6

ADD (cont.)

- setta anche gli altri flag:
  - ZF =1 se tutti i bit valgono zero
  - SF <- MSB(dest)

- In C2, l'MSB rappresenta il segno.

INCREMENT

16 bit

ADD $1, %CX

- **FORMATO**: `INC destination`
- **AZIONE**: Equivale all'istruzione `ADD $1,destination`, con la sola differenza che il contenuto del flag CF non viene modificato.
- **FLAG** di cui viene modificato il contenuto: OF, SF e ZF.

Operandi	Esempi
Memoria	INCB (%ESI)
Registro Generale	INC %CX

1 byte

- **Più compatta**, in quanto nella ADD andrebbe comunque specificata una costante sul numero di bit del destinatario (8, 16, 32)
- Tanti (tanti) anni fa, era anche più veloce (vedremo perché)

SUBTRACT

dest -= source

- **FORMATO**: `SUB source, destination`
- **AZIONE**: Modifica l'operando destinatario sottraendovi l'operando sorgente (il risultato dell'operazione è consistente sia se gli operandi sono interpretati come numeri naturali sia se gli operandi sono interpretati come numeri interi); mette ad 1 il contenuto del flag CF se, interpretando gli operandi come numeri naturali, si è verificato un prestito; mette ad 1 il contenuto del flag OF se, interpretando gli operandi come numeri interi, si è verificato un traboccamento.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria, Registro Generale	SUB 0x00002000,%EDX
Registro Generale, Memoria	SUB %CL,0x12AB1024
Registro Generale, Registro Generale	SUB %AX,%DX
Immediato, Memoria	SUBB $0x5B,(%EDI)
Immediato, Registro Generale	SUB $0x54A3,%AX

SUBTRACT (cont.)

- La SUB calcola `dest -= src`
- Algoritmo indipendente dalla base

- **Prestito uscente** finisce nel CF. Se la SUB opera su numeri **naturali**
  - CF=0 -> la differenza è un naturale
  - CF=1 -> la differenza non è un naturale

SUBTRACT (cont.)

- La SUB calcola anche la differenza di numeri interi
  - Se sono interi, **non guardare CF** (non è significativo)
  - La differenza di numeri concordi è sempre rappresentabile
  - La differenza di numeri discordi è rappresentabile solo se il risultato ha il segno del minuendo
- Tutto questo sta scritto in OF
  - Non è particolarmente importante ricordare l'algoritmo esatto

DECREMENT

- **FORMATO**: `DEC destination`
- **AZIONE**: Equivale all'istruzione `SUB $1,`*`destination`* con la sola differenza che il contenuto del flag CF non viene modificato.
- **FLAG** di cui viene modificato il contenuto: OF, SF e ZF.

Operandi	Esempi
Memoria	DECB (%EDI)
Registro Generale	DEC %CX

ADD WITH CARRY

CF

- **FORMATO**: `ADC source, destination`
- **AZIONE**: Modifica l'operando destinatario sommandovi sia l'operando sorgente sia il contenuto del flag CF (il risultato dell'operazione è consistente sia se gli operandi sono interpretati come numeri naturali sia se gli operandi sono interpretati come numeri interi)
- **FLAG** di cui viene modificato il contenuto: tutti.

Operandi	Esempi
Memoria, Registro Generale	ADC 0x00002000,%EDX
Registro Generale, Memoria	ADC %CL,0x12AB1024
Registro Generale, Registro Generale	ADC %AX,%DX
Immediato, Memoria	ADCB $0x5B,(%EDI)
Immediato, Registro Generale	ADC $0x54A3,%AX

ADD WITH CARRY (cont.)

- Serve a fare le somme di operandi a “molti” bit.
- Esempio: voglio sommare due operandi a 64 bit, ma il sistema accetta solo operandi a 32 bit

ADC ADC CF

ADC	ADD	
56A9C2D4	67A43B5F	+
44B9A5A4	A6B4C55A	=
9B636879	0E5900B9	

64bit rip

+

32

Uso la ADC per i 32 bit più significativi, sommando anche il riporto

Uso la ADD per I 32 bit meno significativi. La ADD potrebbe generare un riporto

CF

SUBTRACT WITH BORROW

- **FORMATO**: `SBB source, destination`
- **AZIONE**: Modifica l'operando destinatario sottraendovi sia l'operando sorgente sia il contenuto del flag CF (il risultato dell'operazione è consistente sia se gli operandi sono interpretati come numeri naturali sia se gli operandi sono interpretati come numeri interi)
- **FLAG** di cui viene modificato il contenuto: tutti.

Operandi	Esempi
Memoria, Registro Generale	SBB 0x00002000,%EDX
Registro Generale, Memoria	SBB %CL,0x12AB1024
Registro Generale, Registro Generale	SBB %AX,%DX
Immediato, Memoria	SBBW $0x255B,(%EDI)
Immediato, Registro Generale	SBB $0x54A3,%AX

SUBTRACT WITH BORROW

dest -= (source + CF)

- è l'istruzione duale dell'ADC, serve per fare le sottrazioni "lunghe".

CF

```
  SBB       SUB
9B636879 0E5900B9 -
56A9C2D4 67A43B5F =
44B9A5A4 A6B4C55A
```

NEGATE

- **FORMATO**: `NEG destination`
- **AZIONE**: Interpreta l'operando destinatario come un numero intero e lo sostituisce con il suo opposto. Qualora l'operazione non sia possibile, mette ad 1 il contenuto del flag OF. Mette ad 1 il contenuto del flag CF eccetto quando l'operando è zero: in tal caso lo mette a 0.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria	NEGB (%EDI)
Registro Generale	NEG %CX

1000 0000

MOV $0x800, %AL

NEGATE (cont.)

NEG %AL

0111 1111

1000 0000

0110

1000+1 1001

- La NEG
  - complementa bit a bit il destinatario
  - aggiunge uno
- Se il destinatario è interpretato come intero (altrimenti **non ha senso**), questo equivale al calcolo dell'opposto.
- l'intervallo di rappresentabilità in C2 non è simmetrico:

$$\left[-2^{N-1}; 2^{N-1} - 1\right]$$

- C'è **overflow** se l'operando è la rappresentazione dell'estremo negativo

COMPARE

- **FORMATO**: `CMP source, destination`
- **AZIONE**: Verifica se l'operando destinatario è maggiore, uguale o minore dell'operando sorgente, sia interpretando gli operandi come numeri naturali che come numeri interi. Aggiorna poi il contenuto dei flag tenendo conto del risultato della verifica (i due operandi rimangono inalterati). L'esatto algoritmo di aggiornamento del contenuto dei flag è noioso da descriversi: l'aggiornamento è comunque consistente con l'interpretazione che del contenuto dei flag sarà data dall'istruzione di salto condizionato (vedi avanti), che in un programma sensato segue sempre l'istruzione CMP.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria, Registro Generale	CMP 0x00002000,%EDX
Registro Generale, Memoria	CMP %CL,0x12AB1024
Registro Generale, Registro Generale	CMP %AX,%DX
Immediato, Memoria	CMPB $0x5B,(%EDI)
Immediato, Registro Generale	CMP $0x45AB54A3,%EAX

COMPARE (cont.)

- La CMP fa quanto segue:
  1) `null = dest - source`
  2) aggiorna i flag come farebbe la SUB
- Unita alle istruzioni di salto condizionato costituisce la base per i costrutti di controllo di flusso.
- Le istruzioni di **salto condizionato** seguono in genere una CMP

CMP ... | CMP %AX, %BX
JGT | JAE

Moltiplicazioni e divisioni

- Sono **diverse** per I naturali e gli interi
  - `MUL, IMUL`
  - `DIV, IDIV`

CF OF

- La **somma** di due numeri a N cifre sta su N (o N+1) cifre
  - Ha senso usare un operando sia come **addendo** che come contenitore del **risultato**

- Il **prodotto** di due numeri a N cifre sta su 2N cifre
  - Fattori e risultato hanno dimensioni non comparabili
  - Non avrebbe senso usare un operando sia come **fattore** che come contenitore del **risultato**

MUL src, dst, ~~DS~~

Moltiplicazione

- Ha sia un **operando** che il **destinatario impliciti**
- Esistono **tre** diversi tipi di moltiplicazione
  - 8 bit: `AX = AL * source`
  - 16 bit: `DX_AX = AX * source` (parte alta su DX, parte bassa su AX)
  - 32 bit: `EDX_EAX = EAX * source` (parte alta su EDX, parte bassa su EAX)

- È il **programmatore** a scegliere, specificando un operando **source** a 8, 16, 32 bit.

PUSH %DX
PUSH %AX
POP %EAX

MUL %BL
MULW (%EDI)

EAX = AX · source

MULTIPLY

- **FORMATO**: `MUL source`
- **AZIONE**: Considera l'operando sorgente come un moltiplicando e l'operando destinatario (implicito) come un moltiplicatore ed effettua l'operazione di moltiplicazione, interpretando gli operandi come numeri naturali.

  Se l'operando sorgente (moltiplicando) è:

  1) ad 8 bit, allora carica in AX il prodotto di AL e source
  2) a 16 bit, allora carica in DX_AX il prodotto di AX e source
  3) a 32 bit, allora carica in EDX_EAX il prodotto di EAX e source

- **FLAG** di cui viene modificato il contenuto: CF, OF: vengono messi a 1 se il risultato non sta sul numero di bit di source, e a zero altrimenti. SF e ZF sono **indefiniti.**

Operandi	Esempi
Memoria	MULB (%ESI)
Registro Generale	MUL %ECX

INTEGER MULTIPLY

- **FORMATO**: `IMUL source`
- **AZIONE**: Considera l'operando sorgente come un moltiplicando e l'operando destinatario (implicito) come un moltiplicatore ed effettua l'operazione di moltiplicazione, interpretando gli operandi come numeri interi.

  Se l'operando sorgente (moltiplicando) è:

  1) ad 8 bit, allora carica in AX il prodotto di AL e source
  2) a 16 bit, allora carica in DX_AX il prodotto di AX e source
  3) a 32 bit, allora carica in EDX_EAX il prodotto di EAX e source
- **FLAG** di cui viene modificato il contenuto: CF, OF, SF (non attendibile) e ZF (non attendibile).

Operandi	Esempi
Memoria	IMULB (%ESI)
Registro Generale	IMUL %ECX

Divisione

- Per questa valgono la maggior parte delle considerazioni fatte per la moltiplicazione, con alcuni problemi in più
  - I risultati sono **due: quoziente e resto**
  - L'operazione può **non essere fattibile** (se il divisore vale 0)

1

- $X\ div\ Y\ \rightarrow Q, R$
- $0 \leq R \leq\ Y - 1$ -> il resto sta sul n. di cifre del **divisore**
- $0 \leq Q \leq X$ -> il quoziente in teoria sta sul n. di cifre del **dividendo**

La divisione nei processori Intel x86

DIV source

- Tre versioni possibili
  - 8 bit: AL = quoziente (AX / source); AH = resto (AX / source)
  - 16 bit: AX = quoziente (DX_AX / source); DX = resto (DX_AX / source)
  - 32 bit: EAX = quoziente (EDX_EAX/source); EDX = resto (EDX_EAX/source)

Y X Q R

source (divisore)	dimensione dividendo	dividendo	quoziente	resto
8 bit	16	%AX	%AL	%AH
16 bit	32	%DX_%AX	%AX	%DX
32 bit	64	%EDX_%EAX	%EAX	%EDX

DIV %BL

Non avevamo detto che $0 \leq Q \leq X$?
In queste divisioni il quoziente sta sulla metà dei bit del dividendo!

AL ← Q(AX/BL)
AH ← R(AX/BL)

- Il programmatore seleziona la versione opportuna, **scegliendo un operando source** (divisore) a 8, 16, 32 bit

La divisione nei processor Intel x86 (cont.)

- Se il quoziente della divisione non sta sul numero di bit previsto, **viene sollevata un'eccezione**
  - La stessa che partirebbe se aveste diviso per zero
  - Il vostro programma **si inchioda**
- **Ci deve pensare il programmatore**, selezionando la versione opportuna.

La divisione nei processor Intel x86 (cont.)

- Voglio dividere 15000 per 3
  - Quale versione uso?

source (divisore)	dimensione dividendo	dividendo	quoziente	resto
8 bit	16	%AX	%AL	%AH
16 bit	32	%DX_%AX	%AX	%DX
32 bit	64	%EDX_%EAX	%EAX	%EDX

```
MOV $3, %CL
MOV $15000, %AX
DIV %CL
```

```
MOV $3, %CX
MOV $15000, %AX
MOV $0, %DX
DIV %CX
```

DIVIDE

- **FORMATO**: `DIV source`
- **AZIONE**: Considera l'operando sorgente come un divisore e l'operando destinatario (implicito) come un dividendo ed effettua l'operazione di divisione, interpretando gli operandi come numeri naturali. Più in dettaglio, compie le azioni che seguono.

i) Se l'operando sorgente (divisore) è:

1) ad 8 bit, allora divide AX per il sorgente, mettendo il quoziente in AL ed il resto in AH
2) a 16 bit, allora divide DX_AX per il sorgente, mettendo il quoziente in AX ed il resto in DX
3) a 32 bit, allora divide EDX_EAX per il sorgente, mettendo il quoziente in EAX ed il resto in EDX

ii) Se il quoziente non è esprimibile su un numero di bit pari a quello del divisore, allora genera un'*interruzione interna* in conseguenza della quale viene messo in esecuzione un opportuno sottoprogramma. Il contenuto dei flag e dei registri destinati a contenere il quoziente ed il resto è, in tal caso, non significativo.

- **FLAG** di cui viene modificato il contenuto: Tutti, ma in modo non attendibile.

Operandi	Esempi
Memoria	DIVB (%ESI)
Registro Generale	DIV %ECX

INTEGER DIVIDE

dividendo
quoziente
resto

divisore

- **FORMATO**: `IDIV source`
- **AZIONE**: Considera l'operando sorgente come un divisore e l'operando destinatario (implicito) come un dividendo ed effettua l'operazione di divisione, interpretando gli operandi come numeri interi [...]
- **FLAG** di cui viene modificato il contenuto: Tutti, ma in modo non attendibile.

Operandi	Esempi
Memoria	IDIVB (%ESI)
Registro Generale	IDIV %ECX

- Speculare alla precedente, lavora su operandi **interi**

Divisione intera – attenzione ai segni

- Nella divisione intera il **resto**
  - **ha sempre il segno del dividendo**
  - è in modulo minore del divisore.
- Ciò vuol dire che il **quoziente viene sempre approssimato all'intero più vicino allo zero** (approssimazione per troncamento)
  - inconsistente con le nozioni di algebra che possediamo
  - bisogna farsene una ragione
- **Esempio:**
- -7 idiv 3: quoziente -2; resto -1
- 7 idiv -3: quoziente -2; resto +1.

Note conclusive su moltiplicazione e divisione

- Scegliere con cura la versione (soprattutto per la divisione).
  - Dipende dal problema e dalle ipotesi che posso fare.
- Ricordare di **azzerare DX o EDX** prima della divisione, se questa è a più di 8 bit
  - Altrimenti i risultati non tornano
- Ricordare che il **contenuto di DX o EDX** viene **modificato** dall'operazione di moltiplicazione/divisione se questa è a più di 8 bit
  - Effetto collaterale difficile da ricordare

→ DIV %BX

Estensione di campo (per gli interi)

- Operazione con cui si rappresenta un numero usando più cifre

025

- Banale per I naturali: aggiungere **zeri in testa**
  - 100110 -> 0100110

- Non può essere lo stesso per gli **interi** (il primo bit dà il segno)
  - 100110 -> 0100110
- Per gli interi, si deve ripetere il bit più significativo
  - 100110 -> 1100110 $-(\overline{110110}+1)$
  - 011011 -> 0011011 $-(\overline{001101}+1)$

CONVERT WORD TO DOUBLEWORD in EAX

- **FORMATO**: `CWDE`
- **AZIONE**: Interpreta il contenuto di AX come un numero intero a 16 bit, rappresenta tale numero su 32 bit e quindi lo memorizza in EAX.
- **FLAG** di cui viene modificato il contenuto: Nessuno

- Esempio: voglio sommare due interi, uno si trova in AX ed uno in EBX.

```
MOV $-5, %AX
MOV $100000, %EBX
CWDE
ADD %EAX, %EBX
```

Istruzioni di traslazione e rotazione

- Servono a variare l'ordine dei bit in un operando destinatario
- Tutte hanno due formati:
  - `OPCODE src, dest`
  - `OPCODE dest`
- src è il **numero di ripetizioni**
  - Immediato oppure registro CL
  - Deve valere al massimo 31
  - Se omesso vale 1

	SX	DX
traslazione	SHL SAL	SHR SAR
rotazione	ROL RCL	ROR RCR

x	x	x					
7	6	5	4	3	2	1	0

SHIFT LOGICAL LEFT

- **FORMATO**: `SHL source, destination`
  `SHL destination`
- **AZIONE**: Interpreta l'operando sorgente come un numero naturale *n* e, per *n* volte, sostituisce il bit contenuto in CF e ciascun bit dell'operando destinatario con il bit che gli è immediatamente a destra, considerando il bit più significativo dell'operando destinatario immediatamente a destra del bit contenuto in CF e sostituendo con 0 il bit meno significativo dell'operando destinatario. ~~Pone infine a 0 il contenuto del flag OF.~~
- **FLAG** di cui viene modificato il contenuto: Tutti.

SHIFT LOGICAL LEFT (cont.)

13 ×
100 = 10^2

1300

SHL $3, %AL

CF

Operandi	Esempi
Immediato, Registro Generale	SHL $1,%EAX
Immediato, Memoria	SHLB $7,0x00002000
Registro CL, Registro Generale	SHL %CL,%BX
Registro CL, Memoria	SHLL %CL,(%EDI)
Memoria	SHLL (%EDI)
Registro Generale	SHL %AX

- Lo shift sinistro *n* volte equivale a **moltiplicare il destinatario per 2^n**
  - Vale in base 2 come in base 10
- Non è segno di grande furbizia usare una MUL quando si può usare una SHL
  - SHL è più veloce, e per un programmatore è estremamente più semplice
- CF viene sovrascritto *n* volte (difficile capire se il risultato è corretto)

1

SHIFT ARITHMETIC LEFT

not
int.

SHL

SAL $3, %AL

- **FORMATO**: `SAL source, destination`

  `SAL destination`

AZIONE: la stessa cosa della SHL

- Sulla dispensa sono riportate alcune differenze minori, ma è un errore (SHL e SAL hanno lo stesso OPCODE sul manuale Intel)

- La SAL moltiplica per 2^{src} un operando **intero**
  - Se MSB cambia valore almeno una volta, OF=1 (risultato non attendibile)

SHIFT LOGICAL RIGHT

- **FORMATO**: `SHR source, destination`
  `SHR destination`
- **AZIONE**: Interpreta l'operando sorgente come un numero naturale *n* e, per *n* volte, sostituisce il bit contenuto in CF e ciascun bit dell'operando destinatario con il bit che gli è immediatamente a sinistra, considerando il bit meno significativo dell'operando destinatario immediatamente a sinistra del bit contenuto in CF e sostituendo con 0 il bit più significativo dell'operando destinatario. Pone infine a 0 il contenuto del flag OF. Il numero naturale *n* non deve superare 31. Se l'operando sorgente non è presente, compie l'operazione una sola volta.
- **FLAG** di cui viene modificato il contenuto: Tutti.

SHIFT LOGICAL RIGHT

32|16 % 100
10^2

Operandi	Esempi
Immediato, Registro Generale	SHR $1,%EAX
Immediato, Memoria	SHRB $7,0x00002000
Registro CL, Registro Generale	SHR %CL,%BX
Registro CL, Memoria	SHRL %CL,(%EDI)
Memoria	SHRL (%EDI)
Registro Generale	SHR %AX

- SHR src, %EAX
  - Corrisponde a dividere EAX per 2^{src}, approssimando il quoziente per difetto
  - Purché EAX sia interpretato come **operando naturale** (vengono inseriti zeri in testa)

SHIFT ARITHMETIC RIGHT

- **FORMATO**: `SAR source, destination`
- `SAR destination`
- **AZIONE**: [...].
- Guardando all'operando destinatario come ad un numero intero, questa istruzione divide tale numero per 2^n e lo sostituisce con il quoziente (approssimato per difetto).

SHIFT ARITHMETIC RIGHT

- La SAR fa **quasi** la stessa cosa della SHR, ma opera sugli **interi**
- Per dividere per due un intero **non si può far entrare zeri a sx**
  - Altrimenti si cambia il segno
  - Si deve ripetere il MSB

- `SAR $3, %AX` divide l'intero contenuto in AX per 2^3=8

Divisioni intere

→ SAR $-1
IDIV

-1 Q = -2

SAR $1, ...

- Attenzione alle **approssimazioni dei quozienti**
  - IDIV: approssima per troncamento (verso lo zero)
  - SAR **approssima sempre a sinistra**
- Quindi SAR e IDIV danno lo stesso quoziente solo quando
  - Il dividendo è **positivo**

  oppure
  - La divisione dà resto nullo

2^n

$$x = \begin{cases} X & X_{N-1} = 0 \\ -\left(\bar{X}+1\right) & X_{N-1} = 1 \end{cases}$$

$$x = -\left(00000010+1\right)_{b2} = -(11)_{b2} = -3$$

Istruzioni di rotazione

- Semplicemente **ruotano i bit**
  - Verso destra / verso sinistra
  - Includendo / non includendo CF nella rotazione
  - Hanno tutte lo stesso formato

ROTATE LEFT, **ROL**

ROTATE RIGHT, **ROR**

ROTATE THROUGH CARRY LEFT, **RCL**

ROTATE THROUGH CARRY RIGHT, **RCR**

ROTATE LEFT

- **FORMATO**: `ROL source, destination`
  `ROL destination`
- **AZIONE**: Interpreta l'operando sorgente come un numero naturale *n* e, per *n* volte, [...]. Il numero naturale *n* non deve superare 31. Se l'operando sorgente non è presente, compie l'operazione una sola volta.
- **FLAG** di cui viene modificato il contenuto: CF, OF.

Operandi	Esempi
Immediato, Registro Generale	ROL $1,%EAX
Immediato, Memoria	ROLB $7,0x00002000
Registro CL, Registro Generale	ROL %CL,%BX
Registro CL, Memoria	ROLL %CL,(%EDI)
Memoria	ROLL (%EDI)
Registro Generale	ROL %AX

Istruzioni logiche

- Applicano gli operatori dell’algebra di Boole (AND, OR, NOT)
- In genere modificano i flag

NOT

- **FORMATO**: `NOT destination`
- **AZIONE**: Modifica l'operando destinatario applicando a ciascuno dei suoi bit l'operazione logica *not*.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Operandi	Esempi
Memoria	NOTL (%ESI)
Registro Generale	NOT %CX

AND

s 0 1 1 0

d 0 1 0 0

- **FORMATO**: `AND source, destination`
- **AZIONE**: Sostituisce ciascun bit dell'operando destinatario con il risultato dell'operazione logica *and* tra il bit stesso ed il corrispondente bit dell'operando sorgente; mette a 0 il contenuto dei flag CF ed OF.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria, Registro Generale	AND 0x00002000,%EDX
Registro Generale, Memoria	AND %CL,0x12AB1024
Registro Generale, Registro Generale	AND %AX,%DX
Immediato, Memoria	ANDB $x5B,(%EDI)
Immediato, Registro Generale	AND $0x45AB54A3,%EAX

OR

1 0 1 0

0 0 1 1

1 0 1 1

- **FORMATO**: `OR source, destination`
- **AZIONE**: Sostituisce ciascun bit dell'operando destinatario con il risultato dell'operazione logica *or* tra il bit stesso ed il corrispondente bit dell'operando sorgente; mette a 0 il contenuto dei flag CF ed OF.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria, Registro Generale	OR 0x00002000,%EDX
Registro Generale, Memoria	OR %CL,0x12AB1024
Registro Generale, Registro Generale	OR %AX,%DX
Immediato, Memoria	ORB $0x5B,(%EDI)
Immediato, Registro Generale	OR $0x45AB54A3,%EAX

EXCLUSIVE OR (XOR)

- **FORMATO**: `XOR source, destination`
- **AZIONE**: Sostituisce ciascun bit dell'operando destinatario con il risultato dell'operazione logica *xor* tra il bit stesso ed il corrispondente bit dell'operando sorgente; mette a 0 il contenuto dei flag CF ed OF.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria, Registro Generale	XOR 0x00002000,%EDX
Registro Generale, Memoria	XOR %CL,0x12AB1024
Registro Generale, Registro Generale	XOR %AX,%DX
Immediato, Memoria	XORB $0x5B,(%EDI)
Immediato, Registro Generale	XOR $0x45AB54A3,%EAX

Utilizzi delle istruzioni logiche

- Istruzioni AND, OR, XOR si usano spesso per lavorare su **singoli bit di registri**, usando un operando sorgente immediato detto **maschera**

- AND: Si usa spesso per testare **singoli bit** di un operando
- Esempio: voglio sapere se il bit n.5 di AL vale uno o zero, posso scrivere:

```
AND $0x20, %AL        #0x20 = 0010|0000 (maschera)
JZ vale_zero          #codice se bit5=0
...                   #codice se bit5=1
```

Utilizzi delle istruzioni logiche (cont.)

- AND si usa per **resettare singoli bit** di un operando
- Esempio: voglio resettare il bit n.6 del registro BH:

```
AND $0xBF, %BH        #0xBF = 1011|1111 (maschera)
```

- OR si usa per **settare singoli bit**, usando un'apposita maschera
- Esempio: voglio settare il bit n.4 del registro CL

```
OR $0x10, %CL         #0x10 = 0001|0000 (maschera)
```

$x \text{ XOR } 0 = x \qquad x \text{ XOR } 1 = \overline{x}$

- XOR si usa per **invertire singoli bit**, usando un'apposita maschera
- Esempio: voglio invertire il bit n.5 del registro AH

```
XOR $0x20, %AH        #0x20 = 0010|0000 (maschera)
```

Utilizzi delle istruzioni logiche (cont.)

- AND si usa per **l'estensione** di operandi naturali
- Voglio sommare due numeri **naturali**, uno in AL, l'altro in EBX

```
MOV $5, %AL
MOV $100000, %EBX
AND $0x000000FF, %EAX
ADD %EAX, %EBX
```

- XOR si usa per **resettare** un registro

```
XOR %EAX, %EAX
```

1 byte

```
MOV $0, %EAX
```

4 byte

≥ 5 byte

operative

Istruzioni di controllo

fetch

JMP etichetta

- Il **flusso del programma** prosegue normalmente in sequenza
- Le istruzioni stanno in memoria consecutivamente
- Il processore preleva un'istruzione, incrementa EIP, la esegue
  - Quindi si eseguono le istruzioni una dietro l'altra

- Tranne quando un'istruzione di controllo **scrive un nuovo valore in EIP**
  - Il flusso sequenziale si interrompe, e si «salta» al nuovo punto
- Istruzioni di **salto** (JMP, Jcon) EIP
- Istruzioni per la gestione di sottoprogrammi (CALL, RET)

JUMP

- **FORMATO**: `JMP %EIP ± displacement`
  `JMP *extended_register`
  `JMP *memory`
- **AZIONE**: Calcola un indirizzo di salto e lo immette nel registro EIP.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

- In Assembler si usa **un nome simbolico** per indicare l'istruzione dove si deve saltare, quindi non è necessario aver presente il formato di indirizzamento (ci pensa l'assemblatore a tradurre in uno dei formati di sopra, in genere il primo)

JUMP if CONDITION MET

- **FORMATO**: `Jcon %EIP ± displacement`
- **AZIONE**: Esamina il contenuto dei flag. Se da questo esame risulta che la condizione *con* è soddisfatta, allora si comporta come l'istruzione `JMP %EIP ± displacement`; in caso contrario termina senza compiere alcuna azione.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

- Vediamo alcuni esempi di condizione

JUMP if CONDITION MET (cont.)

- Condizioni sul valore dei **singoli flag**

CMP
MOV
JZ

JZ	(Jump if Zero) la condizione è soddisfatta se ZF contiene 1; il verificarsi di questa condizione indic che il risultato dell'istruzione precedente è stato zero.
JNZ	(Jump if Not Zero) la condizione è soddisfatta se ZF contiene 0; il verificarsi di questa condizion indica che il risultato dell'istruzione precedente è stato diverso da zero.
JC	(Jump if Carry) la condizione è soddisfatta se CF contiene 1.
JNC	(Jump if No Carry) la condizione è soddisfatta se CF contiene 0.
JO	(Jump if Overflow) la condizione è soddisfatta se OF contiene 1.
JNO	(Jump if No Overflow) la condizione è soddisfatta se OF contiene 0.
JS	(Jump if Sign) la condizione è soddisfatta se SF contiene 1.
JNS	(Jump if No Sign) la condizione è soddisfatta se SF contiene 0.

JUMP if CONDITION MET (cont.)

- Esempi di codice che usa salti con condizioni sui flag

```
ADD %AX, %BX
JC …
ist1
```

Se la somma del contenuto di `AX` e `BX` non è rappresentabile su 16 bit (interpretando i numeri come **naturali**), salta all'indirizzo specificato, altrimenti prosegue da `ist1`.

```
ADD %AX, %BX
JO …
ist1
```

Se la somma del contenuto di `AX` e `BX` non è rappresentabile su 16 bit (interpretando i numeri come **interi**), salta all'indirizzo specificato, altrimenti prosegue da `ist1`.

```
SUB %AL, %BL
JS …
ist1
```

Se la differenza `BL-AL` (interpretando i numeri come **interi**) è un numero negativo, salta all'indirizzo specificato, altrimenti prosegue da `ist1`.

JUMP if CONDITION MET (cont.)

CMP

- Condizioni di **confronto tra numeri naturali**

=	JE	(Jump if Equal) la condizione è soddisfatta se ZF contiene 1; il verificarsi di questa condizione dop l'esecuzione di un'istruzione CMP indica che l'operando destinatario era uguale all'operando sorgent
≠	JNE	(Jump if Not Equal) la condizione è soddisfatta se ZF contiene 0; il verificarsi di questa condizion dopo un'istruzione CMP indica che l'operando destinatario non era uguale all'operando sorgente.
dest > sorc	JA	(Jump if Above) la condizione è soddisfatta se CF contiene 0 e ZF contiene 0; il verificarsi di quest condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggior dell'operando sorgente, essendo gli operandi interpretati come numeri naturali.
≥	JAE	(Jump if Above or Equal) la condizione è soddisfatta se CF contiene 0; il verificarsi di quest condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri naturali.
<	JB	(Jump if Below) la condizione è soddisfatta se CF contiene 1; il verificarsi di questa condizione dop l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore dell'operand sorgente, essendo gli operandi interpretati come numeri naturali.
≤	JBE	(Jump if Below or Equal) la condizione è soddisfatta se CF contiene 1 o ZF contiene 1; il verificar di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario er minore od uguale rispetto all'operando sorgente, essendo gli operandi interpretati come nume naturali.

JUMP if CONDITION MET (cont.)

- Esempi di codice che usa salti con condizioni di confronto tra naturali

```
CMP %AX, %BX
JAE …
ist1
```

Se `BX>=AX` (interpretando i numeri come **naturali**), salta all'indirizzo specificato, altrimenti prosegue da `ist1`.

```
CMP %EDX, %ECX
JB …
ist1
```

Se `ECX<EDX` (interpretando i numeri come **naturali**), salta all'indirizzo specificato, altrimenti prosegue da `ist1`.

- La condizione scritta nell'opcode si riferisce **al destinatario**

JUMP if CONDITION MET (cont.)

- Condizioni di confronto **tra numeri interi**

=	JE	(Jump if Equal) la condizione è soddisfatta se ZF contiene 1; il verificarsi di questa condizione dop l'esecuzione di un'istruzione CMP indica che l'operando destinatario era uguale all'operando sorgent
≠	JNE	(Jump if Not Equal) la condizione è soddisfatta se ZF contiene 0; il verificarsi di questa condizion dopo un'istruzione CMP indica che l'operando destinatario non era uguale all'operando sorgente.
dest > sic	JG	(Jump if Greater) la condizione è soddisfatta se ZF contiene 0 e se il contenuto di SF è uguale a quell di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica ch l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati com numeri interi.
≥	JGE	(Jump if Greater or Equal) la condizione è soddisfatta se il contenuto di SF è uguale a quello di OF; verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operand destinatario maggiore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati com numeri interi.
<	JL	(Jump if Less) la condizione è soddisfatta se il contenuto di SF è diverso da quello di OF; il verificar di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario er minore dell'operando sorgente, essendo gli operandi interpretati come numeri interi.
≤	JLE	(Jump if Less or Equal) la condizione è soddisfatta se ZF contiene 1 oppure se il contenuto di SF diverso da quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CM indica che l'operando destinatario era minore od uguale rispetto all'operando sorgente, essendo g operandi interpretati come numeri interi.

JUMP if CONDITION MET (cont.)

- Esempi di codice che usa salti con condizioni di confronto tra interi

```
CMP %AX, %BX
JGE …
ist1
```

Se `BX>=AX` (interpretando i numeri come **interi**), salta all'indirizzo specificato, altrimenti prosegue da `ist1`.

```
CMP %EDX, %ECX
JL …
ist1
```

Se `ECX<EDX` (interpretando i numeri come **interi**), salta all'indirizzo specificato, altrimenti prosegue da `ist1`.

- La condizione scritta nell'opcode si riferisce **al destinatario**

JUMP if CONDITION MET (cont.)

- Salti condizionati con condizioni di confronto **seguono sempre una CMP** in ogni programma sensato
- Le relazioni di confronto tra naturali **si evincono sempre dai flag**
- Esempio: **jump if below or equal**

```
CMP src, dest
JBE …
```

SUB

- La CMP sottrae src da dest
  - Se dest=src, il risultato è nullo (ZF=1)
  - Se dest<src, la sottrazione genera un prestito (CF=1)
  - JBE quindi testa la seguente condizione: **ZF=1 OR CF=1**

JUMP if CONDITION MET (cont.)

$\mathbb{Z}$

CMP src, dest

JGE

- Anche per le condizioni sugli interi si guardano **i flag**
- Esempio: **jump if greater or equal**
- La CMP sottrae src da dest
  - Se la differenza è rappresentabile, OF=0. In questo caso, affinché dest>=src, il segno del risultato deve essere positivo (SF=0) e nulla
  - Se la differenza non è rappresentabile, OF=1, ed in questo caso il segno del risultato è sbagliato. Quindi, affinché dest>=src, il segno del risultato deve essere negativo (SF=1)
  - JGE quindi salta se è vera la seguente condizione: **OF=SF**

OF = 0 AND SF = 0

OF = 1 AND SF = 1

- Non è necessario ricordarsi le condizioni sui flag. Basta conoscere i codici mnemonici, che sono facili da ricordare

Sottoprogrammi

- Le istruzioni coinvolte sono due:
  - CALL salta ad un sottoprogramma
  - RET ritorna al programma chiamante
- Entrambe fanno riferimento alla **pila**

CALL

JMP

- **FORMATO**: `CALL %EIP ± $displacement`
  `CALL *extended_register`
  `CALL *memory`
- **AZIONE**: Effettua la *chiamata* di un *sottoprogramma*. Più precisamente, salva nella pila corrente il contenuto del registro EIP e poi modifica il contenuto di tale registro in modo del tutto simile a come fa l'istruzione JMP.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

RETURN

- **FORMATO**: `RET`
- **AZIONE**: Effettua il *ritorno* da un sottoprogramma. Più precisamente, rimuove un long dalla pila e con esso rinnova il contenuto di EIP.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

NO OPERATION

- **FORMATO**: `NOP`
- **AZIONE**: Termina senza compiere alcuna azione.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

- Serve a perdere tempo

HALT

- **FORMATO**: `HLT`
- **AZIONE**: Provoca la cessazione di ogni attività del processore.

Il processore alterna la fase di fetch (in cui preleva un'istruzione) con quella di esecuzione dell'istruzione che ha prelevato. Quando preleva un'istruzione HLT, smette di fare qualunque cosa, e si pone in una condizione dalla quale si può uscire solo resettando la macchina.

Protezione / istruzioni privilegiate

⊂ tutte

- Il processore può funzionare in due modalità: **utente** e **sistema**
- In modalità sistema può usare **tutte le istruzioni**
- In modalità utente ne può usare **soltanto alcune**
- I programmi Assembler che scriviamo girano in **modalità utente**
- Tra le istruzioni che non potete utilizzare ci sono quelle **«privilegiate»**
  - HLT
  - IN
  - OUT
- Se le scrivete in un programma, va in esecuzione un'eccezione di protezione, ed il comportamento è dipendente dal sistema

Protezione / istruzioni privilegiate

pippo: JMP pippo

- Non avere la HLT non è un gran problema
- Ma come si fa a fare I/O in Assembler?
  - Si passa attraverso appositi sottoprogrammi di servizio
  - li vedremo più avanti

- Perché le istruzioni di I/O sono privilegiate?
  - Le interfacce sono sistemi **complessi**
  - Facile portarle in stato **inconsistente**
  - I sottoprogrammi di servizio garantiscono un uso corretto delle medesime