

1 Lezione del 24-09-24

1.1 Introduzione

Il corso di reti logiche tratta di:

1. **Linguaggio assembler:** come scrivere programmi semplici, come avviene la compilazione in linguaggio macchina;
2. **Reti logiche:** reti combinatorie, reti combinatorie per l'aritmetica, reti sequenziali asincrone e sincronizzate;
3. **Microprogrammazione:** reti sequenziali sincronizzate, come realizzare una rete logica da specifiche. "Micro" qui sta per *hardware*;
4. **Il calcolatore:** processore, interfacce comuni e convertitori.

1.1.1 Introduzione alle reti logiche

Si parla di reti *logiche* in quanto si guarda all'hardware da una prospettiva funzionale, indipendente dalla sua tecnologia. Ad esempio, una porta NOR sarà implementata con determinati circuiti, ma tutto ciò che interessa a questo corso è come si comporta logicamente: $y = 1 \Leftrightarrow A = B = 0$.

1.2 Programmazione assembly

Il nome corretto del linguaggio sarebbe Assembly, ma noi lo chiameremo Assembler per ragioni storiche. L'assembler è il linguaggio con cui si scrivono le istruzioni eseguite dal processore. Il processore implementa effettivamente un ciclo fetch-execute dove preleva la prossima istruzione macchina (in assembler) dalla memoria e la esegue.

1.2.1 Linguaggio macchina

Il linguaggio macchina (LM) è dato dal contenuto effettivo della memoria che contiene le istruzioni, ergo una sequenza di zero e uno. Il linguaggio assembler adotta una sintassi simbolica per il linguaggio macchina: ad esempio, `MOV %AX, %BX`.

Il processo di trasformazione dall'assembler all'LM si chiama **assemblaggio**, mentre il processo di trasformazione da un linguaggio ad alto livello all'assembler si chiama **compilazione**.

1.2.2 Generalità sull'assembler

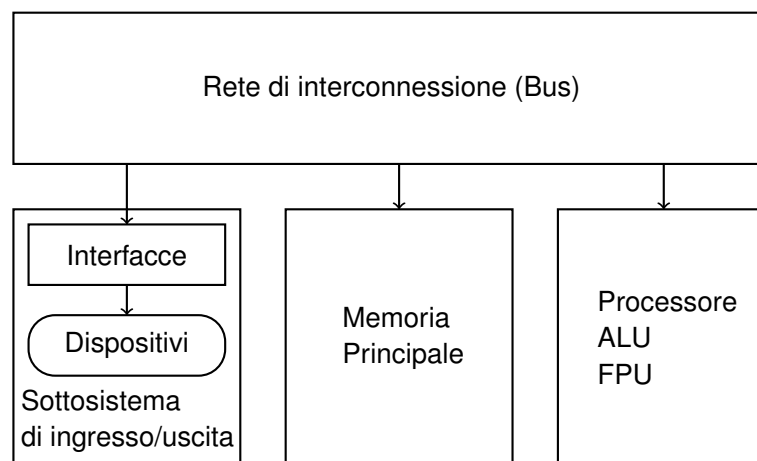
Si dice che assembler è un linguaggio a basso livello. Mancano i costrutti a cui siamo abituati da i linguaggi di alto livello:

1. Non esistono costrutti di flow control (for, if-else, ecc...), tutto si fa con istruzioni di salto.
2. Non esistono tipi variabile: gli operandi sono stringhe di bit che si riferiscono a locazioni in memoria.

Inoltre, l'assembler è strettamente legato all'hardware, ed è specifico per ogni processore. Noi vedremo l'assembler dei processori della famiglia Intel x86, che non è uguale all'assembler dei processori Arm Cortex, ecc... Questo rende il codice in assembler mai portatile. Fatta questa precisazione, possiamo dire che i principi generali restano comunque validi fra famiglie di processori diverse.

Esiste ancora oggi una nicchia di utilizzo del linguaggio assembler: quello dello sviluppo di sistemi embedded. Inoltre, il linguaggio ha un importante significato didattico e culturale.

1.3 Schema a blocchi del calcolatore



Un calcolatore è formato, in linea generale, da una rete di interconnessione (bus) che collega fra di loro:

- Interfacce che comunicano con dispositivi;
- La memoria principale che contiene dati e programmi;
- Il processore, che esegue il ciclo fetch-execute. Possiamo aggiungere che ogni processore, oggi, contiene almeno due blocchi:
 - L'ALU, Arithmetic Logic Unit, che si occupa di calcoli aritmetici su numeri interi (interpretando le stringhe di bit come numeri naturali o interi in complemento a 2) e operazioni logiche;
 - L'FPU, Floating Point Unit, che si occupa dei numeri a virgola mobile.

1.4 Riassunto di rappresentazione dell'informazione

Da qui in poi x è il numero rappresentato e X la sequenza di bit rappresentante.

1.4.1 Numeri naturali

Intervallo di rappresentabilità

n bit rappresentano 2^n naturali sull'intervallo $[0, 2^n - 1]$.

Trasformazione diretta

Per portare un'intero in rappresentazione binaria nel suo corrispondente in base 10, si

sa che presi n bit $b_{n-1}, b_{n-2}, \dots, b_1, b_0$ della rappresentazione X , essi rappresentano il naturale x :

$$x = b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \dots + b_1 \cdot 2 + b_0 = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

Il bit più a sinistra è il Most Significant Bit (MSB), cioè b_{n-1} , quello più a destra il Least Significant Bit (LSD) cioè b_0 .

Trasformazione inversa

Per portare un'intero in base 10 nella sua rappresentazione binaria, si usa l'algoritmo DIV-MOD:

Algoritmo 1 DIV-MOD

Input: x in base 10

Output: X rappresentazione in base 2

Inizializza $q \leftarrow x$, $r \leftarrow 0$, and $i \leftarrow 0$

Crea un'array vuota R per i resti

while $q \neq 0$ **do**

$r \leftarrow q \bmod 2$

Metti r in $R[i]$

$q \leftarrow q/2$

$i \leftarrow i + 1$

end while

Gli $R[n-1], R[n-2], \dots, R[0]$ rimasti (letti al contrario) sono le cifre di X .

1.4.2 Numeri interi in complemento a due

Intervallo di rappresentabilità

n bit rappresentano 2^n interi sull'intervallo $[-2^{n-1}, 2^{n-1} - 1]$.

Trasformazione diretta

Per portare un intero x in base 10 nella sua rappresentazione in complemento a due X su n bit, si decide alternativamente rispetto al segno di x di rappresentare il naturale N in X :

$$N = \begin{cases} x & x \geq 0 \\ 2^n + x & x < 0 \end{cases}, \quad X = N_2$$

dove si nota che nella seconda espressione $2^n + x$ equivale a $2^n - |x|$, dalla negatività di x .

Alternativamente, sui soli numeri negativi:

- Si converte x in rappresentazione binaria.
- Si trova il complemento, ovvero la rappresentazione che inverte tutti i bit (che equivale alla rappresentazione in complemento a 2 dell'opposto -1).
- A questo punto si aggiunge 1, ignorando qualsiasi overflow.

La rappresentazione X trovata è il complemento a 2 di x . Simbolicamente:

$$X = \begin{cases} x_2 & x \geq 0 \\ (\bar{x} + 1)_2 & x < 0 \end{cases}$$

Trasformazione inversa

Per portare la rappresentazione in complemento a due X su n bit di un intero x all'intero stesso, ci si comporta come per le rappresentazioni di naturali, ma prendendo il bit più significativo dagli n bit $b_{n-1}, b_{n-2}, \dots, b_1, b_0$ della rappresentazione X con valenza negativa:

$$x = -b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \dots + b_1 \cdot 2 + b_0 = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

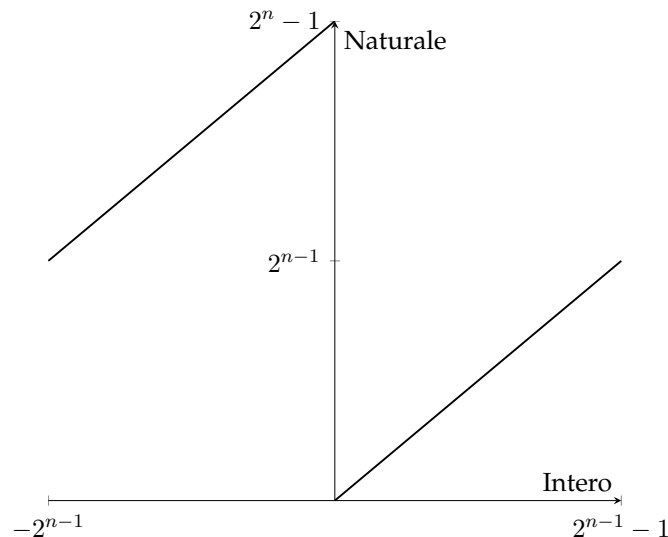
Alternativamente, si nota che il bit più significativo della rappresentazione sarà impostato a 0 per numeri positivi e 1 per numeri negativi. Ciò significa che avremo:

$$x = \begin{cases} X_{10} & X_{n-1} = 0 \\ -(\bar{X} + 1)_{10} & X_{n-1} = 1 \end{cases}$$

dove la barra rappresenta l'operazione complemento.

1.4.3 Rappresentazioni di interi e naturali, diagramma a farfalla

La rappresentazione in complemento 2 su n bit è effettivamente una funzione dal dominio $[-2^{n-1}, 2^{n-1} - 1]$ degli interi al codominio $[0, 2^n - 1]$ dei naturali. Tale funzione prende il nome di *diagramma a farfalla*:



da cui notiamo la relazione fra un'intero e il naturale che lo rappresenta in complemento a 2.

1.4.4 Valori notevoli del complemento a 2

Vale la pena notare alcuni valori notevoli del complemento a 2 su n bit.

- Innanzitutto, 0 rimane 0, ergo una fila di n zeri.
- Uno zero seguito da $n - 1$ uni è il numero più positivo positivo, ergo $2^{n-1} - 1$.
- Aggiungendo uno, si arriva ad un uno seguito da $n - 1$ zeri, che è il numero negativo possibile, ergo -2^{n-1} . Notare che questo combacia col prendere il numero più positivo $2^{n-1} - 1$, e ricavare uno meno del suo opposto -2^{n-1} , che abbiamo appurato essere ciò che accade quando si complementa (e infatti i due numeri sono l'uno il complemento dell'altro).
- Infine, una sequenza di n uno rappresenta il più piccolo numero negativo, ergo -1 .

Si nota che, al pari dei naturali, la rappresentazione dei numeri interi in complemento a 2 è effettivamente ciclica.

1.4.5 Notazione esadecimale

Scrivere lunghe stringhe binarie diventa velocemente complicato. Per questo si adotta una notazione esadecimale per stringhe di 4 bit ($[0, 15]$):

Decimale	Binario	Esadecimale
0	0000	0x0
1	0001	0x1
2	0010	0x2
3	0011	0x3
4	0100	0x4
5	0101	0x5
6	0110	0x6
7	0111	0x7
8	1000	0x8
9	1001	0x9
10	1010	0xA
11	1011	0xB
12	1100	0xC
13	1101	0xD
14	1110	0xE
15	1111	0xF

A questo punto, possiamo denotare qualsiasi stringa binaria come una lista di numeri esadecimali prefissi da 0x (che serve ad indicare la rappresentazione esadecimale stessa), ad esempio 0xC1 (11000001).

1.4.6 Nota sulle potenze di 2

Conviene ricordare le prime potenze di 2:

Esponente	Valore
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024 \approx 1000
11	2048
12	4096
13	8192

e inoltre ricordare che, visto $2^{10} = 1024 \approx 1000$, le unità di misura usuali diventano:

Unità	Potenza
2^{10}	1 KB
2^{20}	1 MB
2^{30}	1 GB

e così via.

1.5 Struttura del calcolatore

1.5.1 Spazio di memoria

La memoria del calcolatore, vista dal programmatore assembler, è uno spazio lineare di 2^{32} (su calcolatori a 32 bit) locazioni (celle) di memoria, dalla capacità di un byte ciascuna. Ogni cella è quindi identificata da un numero di 32 bit, detto **indirizzo**.

Lo spazio di memoria è in larga parte implementato attraverso Random Access Memory (RAM), ovvero memoria volatile. Solo una piccola parte dello spazio è implementata attraverso Read Only Memory (ROM), ovvero memoria permanente, che contiene le istruzioni da eseguire al reset.

Accesso allo spazio di memoria

Il processore può accedere (leggere/scrivere) a:

- Singole locazioni (byte) da 8 bit;
- Doppie locazioni (word) da 16 bit;
- Quadruple locazioni (double word) da 32 bit.

Per gli accessi 16/32 bit si usa l'indirizzo più piccolo delle 2/4 locazioni. Si ricorda che l'indirizzo più grande contiene i bit più significativi.

Gli indirizzi di memoria assembler sono solo simbolici, e vengono tradotti dall'assemblatore, e in parte runtime. Questo significa che non si può accedere a memoria appartenente al sistema operativo, o memoria fuori dai limiti fisici del sistema, ecc...

1.5.2 Spazio di Input/Output

Lo spazio di Input/Output è formato da 2^{16} , ovvero 64k, locazioni o **porte**. Ogni porta ha una capacità di un byte ed è indirizzata da un numero di 16 bit.

Il processore accede alle porte attraverso operazioni particolari di lettura o scrittura (in o out). Spesso le porte sono configurate per un solo tipo di operazione: sola lettura o sola scrittura.

Le locazioni di memoria sono solitamente identiche fra di loro, le porte di I/O no. Indirizzi diversi significano dispositivi diversi, e si rende quindi necessario conoscere fisicamente gli indirizzi.

1.5.3 Processore

Il processore è dotato di una memoria interna formata da locazioni di memoria da 32 bit (**registri**). Questi si dividono in registri **generali**, riservati alle elaborazioni, e **di stato**, riservati a compiti speciali.

Registri generali

I registri iniziano generalmente con la lettera **E**, che sta per *Extended*. Questo perché storicamente i registri erano da 16 bit, e successivamente sono stati estesi a 32 bit. Possiamo quindi riferirci a più sezioni dello stesso registro:

- **EAX**: tutti i 32 bit del registro esteso;
- **AL**: la parte *bassa* del registro **AX**, ergo quella meno significativa, da 8 bit;
- **AH**: la parte *alta* del registro **AX**, ergo quella più significativa, da 8 bit;
- **AX**: il registro **AX** legacy, che combina **AL** e **AH**, da 16 bit.

Alcuni registri vengono storicamente utilizzati per particolari funzioni:

- **EAX** è utilizzato da alcune istruzioni aritmetiche per contenere operandi e risultati. Viene detto **accumulatore**.
- **ESI, EDI, EBX, EBP** vengono detti registri puntatore, dove **B** sta per base e **I** per indice. In particolare:
 - **ESI, EDI** vengono utilizzati come registri indice per accessi in memoria.
 - **EBX** è utilizzato come indirizzo di base per l'accesso in memoria. Viene solitamente detto **base**.
 - **EBP** è utilizzato sempre come indirizzo di base per l'accesso in memoria.
- **ECX** è utilizzato come contatore nei cicli. Viene detto **contatore**.
- **EDX** è utilizzato come operando di operazioni aritmetiche. Viene detto **data**.
- **ESP** è utilizzato per indirizzare la **pila** o **stack**, ovvero una parte di memoria con disciplina LIFO che serve a gestire sottoprogrammi.

Registri di stato

Ricordiamo due registri di stato:

- L'EIP viene detto **instruction pointer**, o **program counter**. Viene usato per contenere l'indirizzo della locazione dalla quale sarà prelevata la prossima istruzione da eseguire. Il contenuto dell'EIP è fissato al reset iniziale, e impostato sulla prima istruzione da eseguire (in memoria ROM) all'indirizzo 0xFFFF0000. Un po' di celle in memoria centrale da questo indirizzo in poi sono implementate in ROM.

Possiamo quindi dire che il ciclo fetch-loop si svolge come segue:

- Il processore preleva dalla memoria, all'indirizzo EIP, una nuova istruzione;
- Incrementa EIP del numero di byte dell'istruzione prelevata;
- Esegue l'istruzione e ripete.

Da questo si ha che le istruzioni in memoria vengono eseguite sequenzialmente nell'ordine in cui sono incontrate, a meno che non si definiscano salti attraverso altre determinate istruzioni.

- L'EF viene detto **extended flag**. Consiste di 32 elementi detti **flag**, fra cui ricordiamo:
 - **OF**: flag di overflow (traboccamento) delle operazioni aritmetiche;
 - **SF**: flag di segno, impostato quando l'ultima operazione restituisce un complemento a 2 con $MSB = 1$ (ergo negativo);
 - **ZF**: flag zero, che viene impostato quando l'ultima operazione restituisce qualcosa di nullo;
 - **CF**: flag di carry (riporto), che viene impostato quando l'ultima operazione richiede un riporto o un prestito.

I flag **OF** e **SF** sono significativi per operazioni su interi. Il flag **CF** è significativo per operazioni su naturali. Il flag **ZF** è significativo per entrambi i tipi di operazione.

Al reset i flag visti finora sono impostati a 0.

2 Lezione del 25-09-24

2.1 Introduzione all'Assembler

2.1.1 Codifica macchina e codifica mnemonica

Possiamo adottare 2 metodi per codificare le istruzioni eseguite dal processore:

- **Codifica macchina**: la serie di zeri e di uni che codificano, nel linguaggio del processore, le operazioni che esegue. Il formato macchina è, nell'architettura che ci interessa, il seguente:

Segmento	Byte	Funzione
I Prefix (Instruction Prefix)	0/1 byte	Usato per modificare l'istruzione
O Prefix (Operand-size prefix)	0/1 byte	Usato per modificare la dimensione degli operandi
Opcode		

- **Codifica mnemonica**: un modo **simbolico** per riferirsi alle istruzioni. Un'istruzione può quindi essere semplicemente: `MOV %EAX, 0x01F4E39`.

Il linguaggio assembler usa la codifica mnemonica delle istruzioni, e dispone di sovrastrutture sintattiche che lo rendono più comprensibile agli umani. Ad esempio, permette l'uso di nomi simbolici per locazioni di memoria: `MOV %EAX, pippo`.

2.1.2 Istruzioni in codifica mnemonica

Un'istruzione ha 3 campi:

- **Codice operativo:** stabilisce quale operazione eseguire;
- **Suffisso di lunghezza:** stabilisce la lunghezza (che può variare) degli operandi;
- **Operandi:** gli operandi su cui si applica l'operazione. Possono essere contenuti in registri, in celle di memoria, nelle porte I/O o direttamente nell'istruzione (**costanti**).

Il suffisso di lunghezza può essere omesso quando è chiaro (essenzialmente quando si usa un registro).

Sintatticamente la struttura è `OPCODEsuffix source, dest`, che diventa qualcosa come `ADD %BX, pluto`. Questa istruzione effettua l'operazione `ADD` (aggiungi), aggiungendo al registro `BX` ciò che è contenuto nel simbolo `pluto`.

Operandi di istruzioni

Le istruzioni ammettono 0, 1 o 2 operandi. Quando sono 2, il primo operando si chiama **sorgente** e il secondo **destinatario**, e solitamente hanno la stessa lunghezza. Quando è 1, l'operando può essere sia sorgente che destinatario a seconda dell'istruzione.

2.1.3 Primo esempio di programma

Si presenta un programma per contare il numero di uno trovati dalla locazione `0x00000100` a `0x0000010i3e` e scriverlo nella locazione `0x00000104`.

```

1 MOVB $0x00, %CL      % sposta $0x00 in %CL
2 MOVL 0x00000100, %EAX % sposta 32 bit da 0x00000100 a %EAX
3 CMPL $0x00000000, %EAX % confronta 32 bit di 0 con il registro %EAX
4 JE   %EIP+$0x07      % salta se uguale a %EIP+$0x07,
5                      % ergo 0x0000020C + 0x07 = 0x00000213
6 SHRL %EAX            % trasla a destra %EAX
7 ADCB $0x00, %CL      % aggiungi a %CL 0 + carry
8 JMP  %EIP-$0x0C      % salta incondizionato a %EIP-$0x0C,
9                      % ergo 0x00000213 - 0x0C = 0x00000207
10 MOVB %CL, 0x00000104 % sposta byte da %CL a 0x00000104

```

Il programma svolge i seguenti passi:

2.1.4 Istruzioni assembler

Le istruzioni assembler si dividono in:

- **Operative:** ovvero quelle che svolgono qualche operazione (`ADD`, `SHR`, `MOV`, `CMP`,);
- **Di controllo:** cioè che si occupano di alterare il flusso del programma (`JMP`, `JE`, ecc...).

Algoritmo 2 Conta 0

```

Inizializza il registro CL (Counter Low) a 0
Sposta i 32 bit da 0x00000000 a 0x00000103 in EAX
while true do
  if EAX è vuoto (tutti zeri) then
    Salta all'ultima istruzione
  end if
  Sposta EAX a destra
  Aggiungi il flag carry (che prende il valore rimosso da EAX) al registro CL
end while
Sposta il byte in CL nella locazione 0x00000104

```

Indirizzamento delle istruzioni operative

Le istruzioni operative si indirizzano attraverso l'**OPCODE** (codice operazione, ADD, MOV, ecc...), seguito da un suffisso (**B**, *byte* da 8 bit, **W**, *word* da 16 bit o **L**, *long* da 32 bit) che può essere omesso, e gli indirizzi sorgente e destinazione.

- Si possono **indirizzare i registri** sia come sorgenti che come destinatari, ovvero gli 8 registri generali da 32 bit, gli 8 registri generali da 16 bit, e gli 8 registri generali da 8 bit (disponibili solo sui registri A, B, C e D). Bisogna precedere i nomi dei registri con %.
- Si può avere **indirizzamento immediato**, ovvero di costanti preceduti da \$, solo sull'operando sorgente.
- Si può **indirizzare la memoria**, ma solo da sorgente o solo da destinatario, specificando un'indirizzo di memoria da 32 bit. Ergo non posso scrivere:

```
1 MOVB pippo, pluto
```

ma devo scrivere:

```
1 MOV pippo, %EAX % qua il suffisso di lunghezza e' implicito
2 MOVL %EAX, pluto
```

L'indirizzamento della memoria, nel caso più generale, è dato da:

$$\text{indirizzo} = \text{base} + \text{indice} \times \text{scala} \pm \text{displacement}$$

dove base e indice sono due registri generali da 32 bit, scala una costante dal valore 1 (default), 2, 4, 8, e displacement una costante intera.

La sintassi è `OPCODEsfx ±disp(base, indice, scala)`.

Si distingue poi l'indirizzamento di tipo:

- **Diretto**, dove si indica soltanto il displacement, che coincide con l'indirizzo. `OPCODEW 0x00002001` significa prendi la word a partire da 0x00002001.
- **Indiretto**, o con registro puntatore, dove si sfrutta un registro: `OPCODEL (%EBX)` significa indirizzare il valore indirizzato da EBX. Si può specificare una scala: `OPCODEL (,%EBX,4)` significa il valore nel registro EBX moltiplicato per 4. Si noti come a essere moltiplicato è l'indice e non la base.

- **Displacement e registro di modifica**, ad esempio da `OPCODEW 0x002A3A2B (%EDI)` si ottiene l'operando a 16 bit ottenuto sommando al displacement `0x002A3A2B` il contenuto di `EDI`, modulo 2^{32} .
- **Bimodificato senza displacement**, ad esempio `OPCODEW (%EBX, %EDI)`, che dipende sia da `EBX` che da `EDI`. Si può anche includere una scala: `OPCODEW (%EBX, %EDI, 8)`.
- **Bimodificato con displacement**, come prima ma con displacement: `OPCODEB 0x002F9000 (%EBX, %EDI)`, ovvero l'indirizzo dato da base in `EBX` + indice in `EDI` + l'offset modulo 2^{32} . Si può avere anche negativo: `OPCODEB -0x9000 (%EBX, %EDI)`, dove si sottrae l'offset invece di sommarlo.

Notare che senza il \$ i numeri in formato esadecimale sono interpretati automaticamente come indirizzi.

- Si possono **indirizzare le porte I/O**, come prima in uno solo dei due operandi. Questo si fa con le istruzioni specifiche `IN` e `OUT`. In particolare si ha indirizzamento di tipo:
 - **Diretto**, solo per indirizzi < 256 , in quanto nel formato macchina ci sono 8 bit. Ad esempio `IN 0x001A, %AL` o `OUT %AL, 0x003A`.
 - **Indiretto con registro puntatore**, usando come registro puntatore soltanto `DX`. Ad esempio `IN (%DX), %AX` o `OUT %AL, (%DX)`.

2.2 Panoramica sulle istruzioni

Abbiamo diviso le istruzioni in **operative** e **di controllo**. Possiamo fare ulteriori suddivisioni:

- **Operative:**
 - Di trasferimento;
 - Aritmetiche;
 - Di traslazione/rotazione;
 - Logiche.
- **Di controllo:**
 - Di salto;
 - Di gestione di sottoprogrammi.

Conviene definire formato, funzionamento, comportamento sui flag e modalità di indirizzamento ammesse per gli operandi di ogni operazione, in quanto l'assembler non è **ortogonale**, ergo ci sono particolari restrizioni su *quali* operandi e modalità di indirizzamento possono essere combinate.