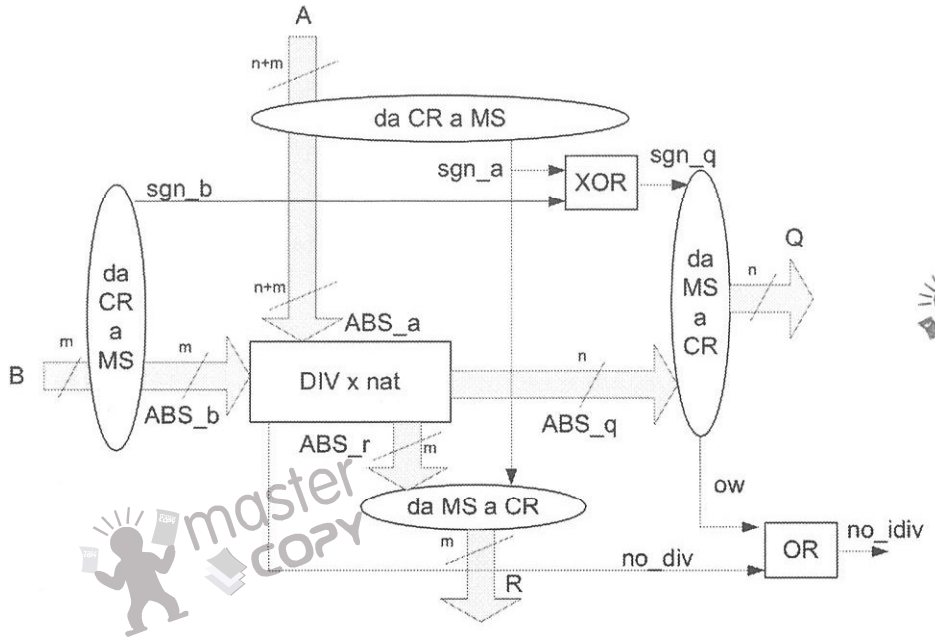
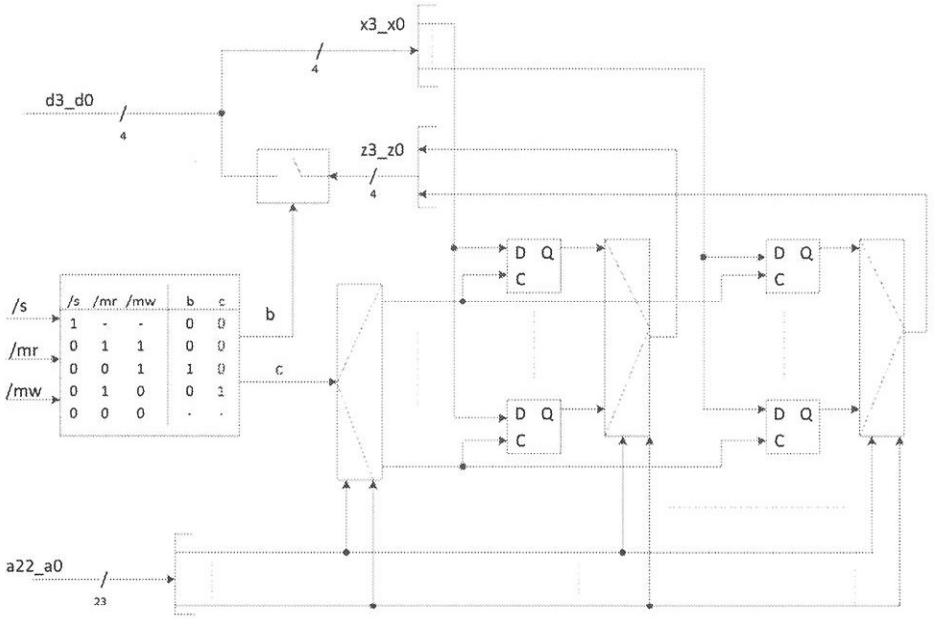
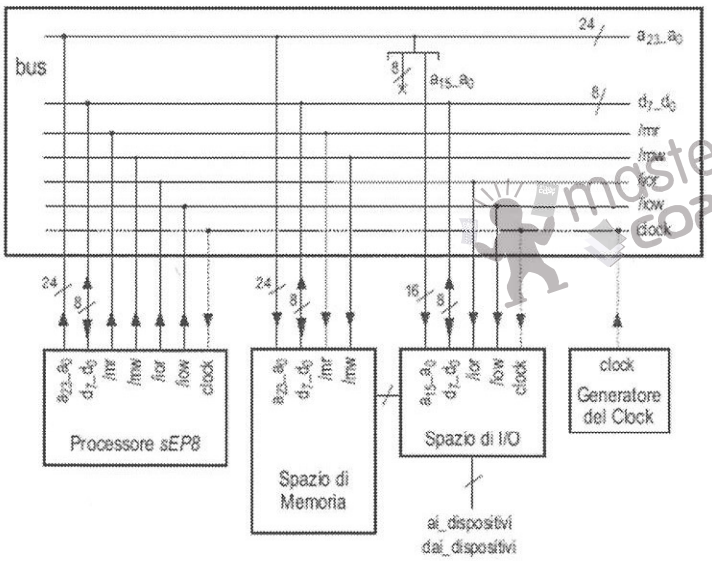






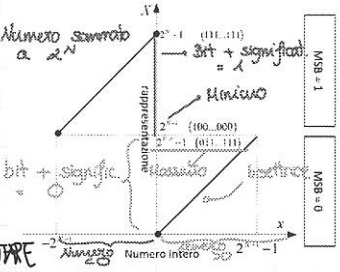
RETI LOGICHE

Programmazione Assembler

- **INTRODUZIONE**: Linguaggio in cui si scrivono le istruzioni nello stesso modo in cui vengono ESEGUITE dal processore
  - ↳ preleva le istruzioni dalla memoria e le esegue
  - Memoria (linguaggio macchina): Sequenze di 0 e 1 ⟶ Assembler = sintassi simbolica
    - ↳ la sequenza viene tradotta dall'ASSEMBLATORE (1:1) ⟶ ≠ da COMPILATORE
      - C++: alto livello
      - Assembler: basso livello
- **STRUTTURA di BASE**: mancano costrutti di flusso (esiste solo go to) e mancano i Tipi di VARIABILE ⟶ l'interpretazione sta al programmatore
  - PROCESSOR-SPECIFIC: le istruzioni VARIANO in base al tipo di processore
    - ↳ Studiamo l'assembler per Intel x86
  - UTILIZZO: Sistemi embedded, elettrodomestici, sistemi IoT
    - ↳ IMPORTANZA: fa capire il funzionamento del processore



- **CALCOLATORE**
  - ⟶ Sottosistema di I/O
    - I: codifica in sequenze di bit quello che noi forniamo in ingresso
    - O: il sistema traduce i bit in caratteri, immagini, etc
    - Dispositivi ed interfacce ⟶ Permettono di interagire allo stesso modo con diversi dispositivi
  - ⟶ Memoria principale: contiene le istruzioni che vengono eseguite
  - ⟶ Processore
    - ALU: arithmetic logic unit = operazioni aritmetiche e logiche
    - FPU: floating point unit = calcoli in virgola mobile
    - ↓
    - La circuiteria smista automaticamente le istruzioni
- **La ALU**: operazioni logiche (AND, OR, NOT) su bit e operazioni aritmetiche su numeri naturali in base 2 e numeri interi in C_2.
  - ⟶ NATURALI: su N bit l'intervallo è $[0, 2^N-1]$
    - ↳ Corrispondenza: $b_{N-1}, b_{N-2}, \dots b_1, b_0 \longleftrightarrow X = \sum_{i=0}^{N-1} b_i \cdot 2^i$
  - ⟶ INTERI C_2: su N bit l'intervallo è $[-2^{N-1}, 2^{N-1}-1]$
    - ↳ Corrispondenza: $X = \begin{cases} x & x \geq 0 \\ 2^N + x & x < 0 \end{cases}$ oppure $X = |x|_{2^N}$
    - ↓
    - Il bit più significativo DETERMINA il SEGNO
    - ↳ $x = \begin{cases} X & \text{se } X_{N-1} = 0 \\ -(\overline{X}+1) & \text{se } X_{N-1} = 1 \end{cases}$ dove $\overline{X}$ è il COMPLEMENTARE



  - ⟶ NOTAZIONE ESADECIMALE: 4 bit rappresentano sempre un numero tra 0 e 15
    - ↳ Si utilizzano le cifre da 0 a 9 per le stringhe corrispondenti e A, B, C, D, E, F per i numeri da 10 a 15
    - ↓
    - Prefisso 0x e poi divisione da destra verso sinistra
- **STRUTTURA del CALCOLATORE**
  - ⟶ 3 SPAZI: 1. Memoria: 2^{32} celle contigue di grandezza 1 byte identificate da un numero a 32 bit (indirizzo) ⟶ 2^{32} byte = 4 GB
    - ↓
    - La CPU accede in R/W a
      - Singola locazione: 8 bit
      - Doppia locazione: 16 bit } utilizzano l'indirizzo più piccolo
      - Quadrupla locazione: 32 bit
      - ↳ L'indirizzo più grande contiene i bit più SIGNIFICATIVI

Esempio:

0x1C	0x 3F65B432
0x3A	0x 3F65B432
0x A2	0x 3F65B432
0x C6	0x 3F65B432

8 bit: 0x 3F65B432 ⟶ 0x1C
16 bit: 0x 3F65B432 ⟶ 0x 3A1C
32 bit: 0x 3F65B432 ⟶ 0x C6 A2 3A1C

↓ Maggior parte → Celle EQUIVALENTI

RAM (Random acces memory): Memoria VOLATILE ⟶ Al reset del processore il contenuto della RAM è CASUALE

⟶ La ROM contiene le istruzioni che il processore esegue alla partenza

⟶ Come posso sapere se posso riferire un indirizzo? (Memoria non presente oppure già occupata) ⟶ Quando si scrive in assembler gli indirizzi vengono riferiti in maniera simbolica, quindi l'assemblatore si occupa della correttezza

2. I/O: 2^{16} locazioni / porte di 1 byte di capacità indirizzabile da indirizzi di 16 bit
   ↳ Il processore ci accede in r/w e una porta in genere ha un solo metodo di accesso in quanto rappresenta un'INTERFACCIA e quindi le celle NON SONO IDENTICHE

3. Processore: Il programmatore vede una serie di REGISTRI, ovvero locazioni di memoria di 32 bit
   ↳ GENERALI: Elaborazione
   ↳ DI STATO: Avanzamento della computazione

master COPY COPISTERIA

- REGISTRI IMPORTANTI:

GENERALI ⟶ EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP Esempio EAX ⟶ 

⟶ E = extended: Ora i registri sono a 32 bit. I 16 bit bassi sono riferibili autonomamente

⟶ possono essere riferiti anche pezzi di 8 bit: xH (AH, BH) ⟶ H = high, x = low della parte "senza E"

NOMENCLATURA relativa a FUNZIONI: EAX ⟶ A = accumulatore: operazioni aritmetiche, contengono operandi e risultati

ESI, EDI EBX, EBP ⟶ puntatori: Assembler consente di utilizzare 2 indici insieme

ESP ⟶ S = stack: utilizzati per indirizzare la pila o lo stack utilizzata per sottoprogrammi ⟶ Punta a top della pila

EBX ⟶ B = base: accesso in memoria

ECX ⟶ C = counter: contatore nei cicli

EDX ⟶ D = data: operazioni aritmetiche

ESI ⟶ source index

EDI ⟶ destination index

EBP ⟶ base pointer: registro base per accessi in memoria

master COPY COPISTERIA

DI STATO

EIP: (program counter) ⟶ contiene l'indirizzo della prossima istruzione da prelevare

↓ Quando il processore parte deve contenere un valore significativo: punta a ROM

FUNZIONAMENTO del PROCESSORE: guarda EIP, va in memoria all'indirizzo scritto in EIP, la preleva, incrementa EIP tanto quanto è grande l'istruzione e la esegue

↓ Alterazione del funzionamento

↓ Il flusso di esecuzione è SEQUENZIALE

Buono esempi istruzioni di STATO: scrivono nuovi valori in EIP in modo da eseguire altre istruzioni

EF: registro dei flag a 32 bit tutti SIGNIFICATIVI ⟶ segnalano una condizione (condizioni)

⟶ OF (Overflow flag): Vale 1 se durante l'esecuzione dell'ultima istruzione si è avuto un overflow: risultato di operazione su interi NON RAPPRESENTABILE

⟶ SF (Sign flag): è 1 se l'ultima istruzione ha generato un risultato con MSB = 1. Se il numero è intero, allora sarà NEGATIVO.

⟶ ZF (Zero flag): è 1 se l'ultima istruzione ha generato un risultato con tutti zeri

⟶ CF (Carry Flag): è 1 se l'ultima istruzione ha generato un riporto ⟶ NATURALI: errore

DA CONTROLLARE: Operazioni su INTERI: OF, SF, ZF

Operazioni su NATURALI: CF, ZF

Al reset ⟶ EIP vale 0x FFFF0000 ⟶ La memoria a questo indirizzo deve essere ROM

⟶ EF vale 0 (tutti i flag)

- **CODIFICA MACCHINA e MNEMONICA**
  - → Macchina: serie di 0 e 1 che codificano le istruzioni eseguite dal processore
  - → Mnemonica: istruzioni scritte in codice ASCII tradotte poi dall'assemblatore con traduzione 1:1 (1 riga = 1 istruzione)
    - ↓ Linguaggio Assembler: codifica mnemonica + scorciatoie sintattiche che semplificano l'utilizzo (ad esempio i nomi assegnati alle celle di memoria)

Esempio

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

→ Formato macchina

↓ Diventa

MOV % EAX, 0x01F4E39A : Indirizzo di memoria specificato dall'utente

MOV % EAX, pippo : l'assemblatore mappa automaticamente la memoria

- **SINTASSI GENERICA delle ISTRUZIONI**: OPCODE suffix source, dest
  - 3 CAMPI → Codice operativo: operazione da eseguire
  - → Suffissi di lunghezza: lunghezza degli operandi
  - → Operandi: Registri, celle di memoria, porte di I/O, nell'istruzione

ESEMPIO: ADD % BX, pluto : Somma il contenuto di BX e lo mette nell'indirizzato pluto

Le istruzioni ammettono 0, 1, 2 operandi → Nell'ultimo caso il primo è quello sorgente (sorgente o destinatario) che non viene modificato, mentre il secondo è il destinatario che riceve il risultato → Sono della STESSA LUNGHEZZA

- **PRIMO PROGRAMMA di ESEMPIO**: preleva dalla memoria un operando a 32 bit, che si trova all'indirizzo 0xCCCCC100, conta il numero di bit a 1 nell'operando e scrive il risultato nella locazione di memoria di indirizzo 0xCCCCC104

```
Indirizzo    Contenuto
0x00000100
0x00000101
0x00000102
0x00000103
0x00000104
...
0x00000200   MOVB $0x00, %CL      → scrive 0 in CL
0x00000202   MOVL 0x00000100, %EAX → copia il contenuto delle celle indicate in EAX
0x00000207   CMPL $0x0000000, %EAX → Confronta la costante 0 su 32 bit con EAX e scrive nei flag
0x0000020A   JE %EIP+$0x07       → Se la condizione è verificata salta ad EIP + 7
0x0000020C   SHRL %EAX           → Trasla a destra i bit di EAX
0x0000020E   ADCB $0x00, %CL     → Somma 0 + CF in CL
0x00000211   JMP %EIP-$0x0C      → Salta all'istruzione in EIP - 0C
0x00000213   MOVB %CL, 0x00000104 → Termina se verificata la condizione
0x00000218   ...
```

(Inizio Programma; Src; Dest; costante 0 su 8 bit)

- **TIPOLOGIA di ISTRUZIONI in ASSEMBLER**
  - istruzioni operative: compiono azioni sugli operandi (svolgono qualche compito)
  - istruzioni di controllo: alterano il normale (sequenziale) flusso di esecuzione del programma
- **Le ISTRUZIONI OPERATIVE e le modalità di INDIRIZZAMENTO**

OPCODE size source, destination → individuati con modalità di indirizzamento

↳ B = byte, W = word, L = long (double word) : 8, 16, 32 bit → Utili quando vengono usati registri

MODALITÀ di INDIRIZZAMENTO

1. **REGISTRO**
   - → 8 registri generali a 8 bit
   - → 8 registri generali a 16 bit
   - → 8 registri generali a 32 bit

   → Nome del registro preceduto dal simbolo % e può essere usato sia per l'operando sorgente che per il destinatario

   ESEMPI: OPCODE %di
   OPCODE % EAX, % EBX
   OPCODE %AH, % CL
2. **IMMEDIATO** costante all'interno dell'operando sorgente mettendo $ prima dell'indirizzo

   ESEMPI: OPCODE $0x20, %AL
   OPCODE $0x5923A20B, % ECX
3. **DI MEMORIA**: O il sorgente, o il destinatario, ma non entrambi, possono essere operandi in memoria

Caso generale: indirizzo = | base + indice * scala ± displacement | modulo 2^{32} → Prende i primi 32 LSB
(costante intera)

↓ SINTASSI

Registro generale a 32 bit → 1 (default), 2, 4, 8 → Serve per accedere vettoriali

OPCODEsfx ± disp (base, indice, scala) Registri B per base, registri I per indice

↓

TIPOLOGIE di INDIRIZZAMENTO di MEMORIA

INDIRIZZAMENTO DIRETTO: si usa SOLTANTO il displacement che coincide con l'indirizzo
↳ OPCODEW 0x0002C001 : il suffisso è NECESSARIO (non è un registro)

INDIRIZZAMENTO INDIRETTO: utilizza un registro puntatore

ESEMPI: OPCODEL (%EBX) → EBX è considerato come REGISTRO BASE
OPCODEL (, %ESI, 4) → ESI è considerato come REGISTRO INDICE

INDIRIZZAMENTO CON DISPLACEMENT e REGISTRO DI MODIFICA

ESEMPIO: OPCODEW 0x002A3A2B (%EDI)

INDIRIZZAMENTO BIMODIFICATO SENZA DISPLACEMENT

ESEMPI: OPCODEW (%EBX, %EDI)
OPCODEW (%EBX, %EDI, 8)

INDIRIZZO BIMODIFICATO CON DISPLACEMENT

ESEMPI: OPCODEB 0x002F9000 (%EBX, %EDI)
OPCODEB -0x9000 (%EBX, %EDI)

INDIRIZZAMENTO delle PORTE di I/O

Uno dei due operandi può trovarsi nella porta di I/O. Per l'indirizzamento usiamo IN / OUT

INDIRIZZAMENTO DIRETTO: si riporta l'indirizzo direttamente negli 8 bit dell'I/O per scrivere un indirizzo < 256

INDIRETTO CON INDIRIZZO PUNTATORE: Si può utilizzare SOLO il registro DX

ESEMPI:
```
IN  0x001A, %AL
IN  (%DX), %AX
OUT %AL, 0x003A
OUT %AL, (%DX)
```

NOTE SULLA SINTASSI

a. Le costanti vanno precedute da $
b. I registri vanno preceduti da %
c. In un'istruzione, un numero non preceduto da $ è un INDIRIZZO
d. SUFFISSI: B = byte, 8 bit; W = word, 16 bit; L = long, 32 bit → può essere OMESSO se l'operando è indirizzato con indirizzamento di registro

La GESTIONE della MEMORIA tramite la PILA

Zona di memoria gestita con la politica LIFO (Last IN, First out). La memoria è gestita tramite una PILA perché il calcolatore funziona tramite sottoprogrammi
↳ vi sono delle istruzioni che chiamano i sottoprogrammi che poi ritornano al programma principale



→ L'esecuzione non è sequenziale

↓

Alla chiamata viene fatta una push dell'indirizzo di ritorno. Quando il sottoprogramma ritorna fa una pop e recupera l'indirizzo → è possibile ANNIDARE e scrivere programmi RICORSIVI

IMPORTANTE

La PILA serve come memoria temporanea perché i registri generali sono pochi ed alcuni sono riservati ad alcune operazioni. "Parcheggio" quindi i dati nella pila

