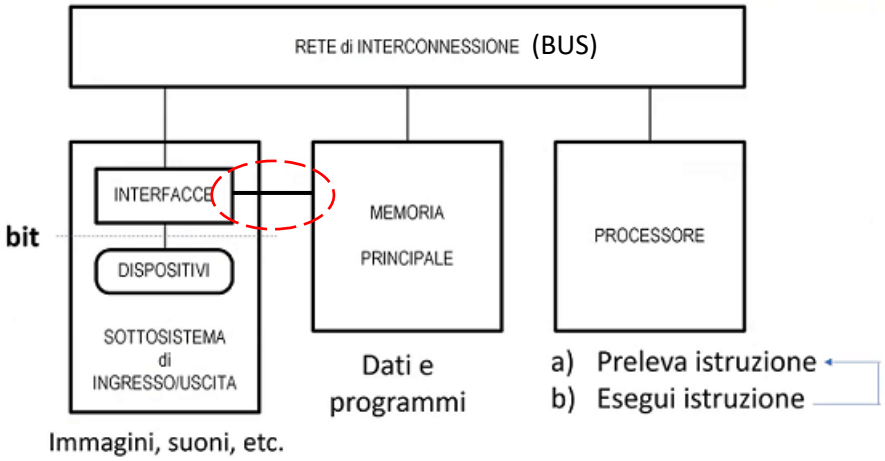
Struttura del calcolatore

Il blocco finale consiste nel descrivere in Verilog un calcolatore, TUTTO, da cima a fondo. Riprendiamo la solita immagine del calcolatore di Von Neumann



Descriveremo processore, memoria, interfacce e dispositivi di I/O utilizzando quanto studiato fino ad ora! Chiaramente il calcolatore che descriveremo non sarà quello che normalmente utilizziamo per seguire le lezioni (quello è troppo complicato), ma un qualcosa che fino a trent'anni fa (quando Stea studiava) poteva essere considerato piuttosto potente.

Cosa possiamo dire sui vari moduli?

- **Sottosistema di I/O**. Si gestisce la codifica delle informazioni ed il loro scambio col mondo esterno (in entrambi i sensi). All'interno abbiamo:
  - **Dispositivi** (trasduttori), che effettuano la codifica vera e propria;
  - **Interfacce**, che gestiscono i vari dispositivi, cioè standardizzano il colloquio tra processore e trasduttore (necessario, il processore non deve conoscere i trasduttori per poter lavorare).

    Ciascuna interfaccia possiede un numero piccolo di registri dove il processore può svolgere operazioni di lettura o di scrittura (in casi rari entrambe): un registro in un interfaccia può contenere, ad esempio, l'ultimo tasto premuto della tastiera, oppure può indicare se deve essere accesa una spia del nostro dispositivo.

- **Memoria principale**. Contiene le istruzioni da eseguire e i dati elaborati dal processore (ricordiamo che alcuni dati possono essere ospitati nel sottosistema di I/O). Una parte di memoria è adibita a memoria video: non la faremo, ma dobbiamo tenere conto del collegamento diretto tra interfaccia e memoria principale (devo salvare in memoria la replica di ciò che stampo sullo schermo). La memoria è realizzata con tecnologia RAM, e in parte ROM (o EPROM, per eseguire certe istruzioni al reset).

- **Processore**. Il processore ciclicamente preleva istruzioni e le esegue. Normalmente abbiamo una sequenza di istruzioni eseguita nell'ordine posto, salvo utilizzo di istruzioni operative che alterano il normale flusso (si indica un nuovo indirizzo e il prelievo di istruzioni riparte da lì). Il processore si ferma quando incontra un'istruzione *HLT*.

  Al momento del reset il processore deve essere avviato in modo consistente.
  - Devo indicare la prima operazione da leggere (un'operazione ben precisa)
  - Chiaramente l'indirizzo di memoria ci porta a un'area realizzata con tecnologia ROM/EPROM, quindi una memoria non volatile.

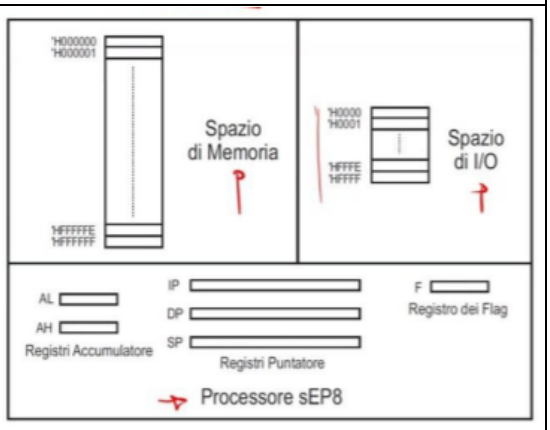
  Posso fare questo inizializzando l'IP (*Instruction pointer*) e altri registri. La memoria non volatile contiene al suo interno un programma bootstrap eseguito all'accensione del calcolatore.

  Quale processore utilizzeremo? Il sEP8 (acronimo di *8-bit simple Educational processor*), che presenta le seguenti proprietà:
  - elabora dati a 8 bit;
  - lavora in aritmetica in base 2 con interi rappresentati in C2;
  - indirizza memoria di 16Mbyte (quindi avrò 24 fili per rappresentare tutti gli indirizzi).

Il calcolatore consiste sostanzialmente in un insieme di RSS: tutti gli elementi che vedremo, tranne dispositivi (parti elettromagnetiche che non sono di nostro interesse) e parte della memoria (che è una RSA), sono RSS. Tutte le RSS presenteranno un piedino di reset e saranno collegate allo stesso circuito di reset (in modo tale da avere un avvio consistente).

In particolare, descriveremo il processore come una RSS sintetizzabile in Parte Controllo e Parte operativa. Ne daremo una specifica come con una qualunque RSS. Oltre a questo vedremo come il processore si interfaccia con le altre reti e qual è il suo comportamento osservabile (il suo linguaggio macchina, perderemo un po' di tempo sulla questione).

Calcolatore visto dal programmatore



- La memoria consiste in uno spazio lineare di 2^{24} byte (16Mbyte)
- Gli indirizzi sono a 24bit (combinazioni possibili di valori sono, non a caso, 2^{24})

- Lo spazio di I/O (registri di interfaccia) consiste in uno spazio lineare di 2^{16} locazioni o porte da un byte.
- Gli indirizzi sono a 16bit (tenerne conto quando vediamo il bus).
- Queste locazioni consistono nell'insieme dei registri di interfaccia che il processore può teoricamente indirizzare. Perché teoricamente? Non è detto che ad ogni locazione corrisponda un registro di interfaccia, anzi è molto probabile che la maggior parte di questo spazio non presenti implementazione fisica (le interfacce sono poche).
- Il processore, da queste porte, può leggere o scrivere un byte alla volta (quindi se dobbiamo leggere più byte dovremo svolgere più operazioni)

- **Registri del processore**:
  - Registri accumulatore (AH, AL, 8 bit), contengono operandi di elaborazioni.
  - Registro dei flag (8 bit), i 4 bit più interessanti per noi sono il CF, lo ZF, il SF e l'OF.
  - Registri puntatore (a 24 bit, visto che devono contenere indirizzi):
    - IP (Instruction pointer), indirizzo della prossima istruzione da eseguire;
    - SP (Stack pointer), contiene l'indirizzo del top della pila;
    - DP (Data Pointer), contiene l'indirizzo di operandi a seconda delle modalità di indirizzamento.

  Al reset dobbiamo inizializzare il registro dei flag e l'instruction pointer, nel seguente modo

```
F <= 'H00;
IP <= 'HFF0000; // Chiaramente memoria ROM implementata a partire da questo indirizzo)
```

  Osservazione: abbiamo anche altri registri, noi abbiamo visto solo i registri utilizzabili nelle istruzioni.

Linguaggio Assembler e Linguaggio macchina

- All'inizio del corso abbiamo introdotto le peculiarità del linguaggio Assembler, che è legato al linguaggio macchina da un rapporto 1:1. Per descrivere il calcolatore dovremo parlare di linguaggio macchina, in modo da poter descrivere il comportamento osservabile della macchina.

- Il linguaggio Assembler, ricordiamo, è diverso da processore a processore. In questo caso utilizzeremo un linguaggio Assembler simile il più possibile a quello già visto per i processori Intel. Chiaramente l'obiettivo non è programmare in Assembler, ma disporre il linguaggio in modo tale che il calcolatore sia facilmente descrivibile.

Linguaggio Assembler

- **Formato delle istruzioni:**

```
OPCODE source, destination
```

  Dove OPCODE è il codice operativo dell'istruzione, mentre source e destination indicano, rispettivamente, l'operando sorgente e quello destinatario. In alcuni casi è assente l'operando source, in altri (rari) entrambi gli operandi (*NOP* e *HLT*).

- **Modalità di indirizzamento**:
  - Indirizzamento di registro, uno o entrambi gli operandi sono nomi di registro
    ```
    OPCODE AL, AH
    OPCODE DP
    ```
  - Indirizzamento immediato, l'operando sorgente è specificato direttamente nell'istruzione come costante (chiaramente possiamo fare questa cosa solo con l'operando sorgente)
    ```
    OPCODE $0x10, AL
    ```
  - Indirizzamento di memoria, valido per il sorgente o per il destinatario (mai contemporaneamente). L'indirizzamento di memoria può essere
    - Diretto, l'indirizzo è specificato direttamente nell'istruzione
      ```
      OPCODE  0x1010, AL
      ```
    - Indiretto, la locazione di memoria ha indirizzo contenuto nel registro DP
      ```
      OPCODE (DP), AL
      ```
  - Indirizzamento delle porte di I/0: le porte di I/O si indirizzano in modo diretto, specificando l'indirizzo della porta nell'istruzione stessa
    ```
    IN 0x1010, AL
    OUT AL, 0x9F10
    ```

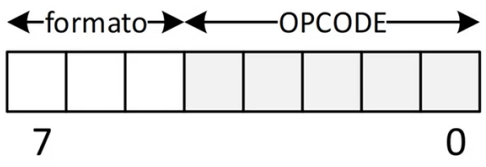
- **Istruzioni di controllo**: istruzioni che alterano il flusso di esecuzione del programma. Permettono salti, condizionati e non, chiamate di sottoprogramma ed istruzioni di ritorno
  ```
  JMP indirizzo
  Jcon indirizzo
  CALL indirizzo
  RET
  ```
  Nelle prime tre istruzioni si specifica l'indirizzo a cui si salta (si sostituisce l'indirizzo di IP), le ultime due interagiscono con la pila. La CALL salva in pila il contenuto di IP (3 BYTE), cioè l'indirizzo dell'istruzione successiva alla CALL. La RET preleva dalla pila un indirizzo (3 byte) e lo sostituisce ad IP.

Linguaggio macchina e formati

- Un processore deve tradurre un'istruzione Assembler
  ```
  OPCODE source, destination
  ```
  in una sequenza di zeri e uni con una certa sintassi. La sintassi è il linguaggio macchina del processore, che deve essere compatta e facile da interpretare (per il compilatore, non per noi).
  - La prima cosa che guardano gli esseri umani, normalmente, è il tipo di operazione (l'Assembler, concepito per gli esseri umani, pone il tipo prima degli operandi).
  - I processori, invece, guardano per prima cosa gli operandi. Vediamo degli esempi
    - `MOV AH, AL`
      Gli operandi sono già posti all'interno di registri
    - `MOV $0x10, AL`
      Il processore deve leggere in memoria l'operando sorgente indicato nell'istruzione
    - `MOV (DP), AL`
      Il processore dovrà leggere in memoria per procurarsi l'operando sorgente. L'indirizzo non è il registro DP, ma il valore contenuto nel registro stesso.

    Dobbiamo distinguere, in queste operazioni, la fase di fetch dalla fase di esecuzione: la prima consiste nel procurarsi gli operandi (e può essere diversa in base all'indirizzamento scelto), la seconda è uguale per tutte queste istruzioni (cioè spostare valori)

- Ciascuna istruzione macchina è lunga almeno un byte. Il primo byte di ogni istruzione codifica sia il **tipo di operazione** (su 5 bit, 32 opcode possibili) che il modo in cui si devono recuperare gli operandi (su 3 bit, 8 formati possibili). Quest'ultima cosa è detta **formato dell'istruzione**.

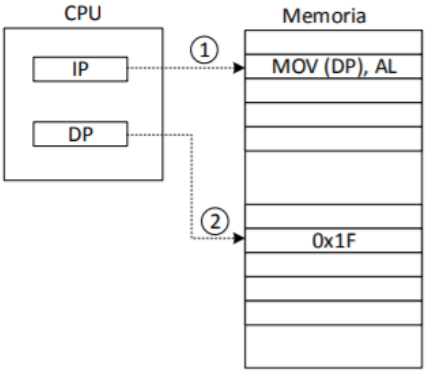


- **Formati possibili**:

  - Formato F0 (000)
    - Categoria in cui rientrano tutte le istruzioni per le quali non è necessario compiere nessuna azione per procurarsi gli operandi.
    - In questa categoria rientrano operazioni in cui gli operandi sono registri o dove non sono presenti operandi (HLT, NOP, RET).
    - La fase di fetch consiste esclusivamente nella lettura di un byte (quello dell'istruzione) visto che non c'è altro da fare.

```
HLT         |00000000|
NOP         |00000001|
MOV AL,AH   |00000010|
MOV AH,AL   |00000011|
INC DP      |00000100|
SHL AL      |00000101|
SHR AL      |00000110|
NOT AL      |00000111|
SHL AH      |00001000|
SHR AH      |00001001|
NOT AH      |00001010|
PUSH AL     |00001011|
POP AL      |00001100|
PUSH AH     |00001101|
POP AH      |00001110|
PUSH DP     |00001111|
POP DP      |00010000|
RET         |00010001|
```

  - Formato F2 (010)
    - Categoria in cui rientrano le istruzioni dove l'operando sorgente si trova in memoria ed è indirizzato tramite DP.
    - Il sorgente deve essere ripescato in memoria. Dovrò fare una seconda lettura in memoria per portare l'operando sorgente dentro il processore.

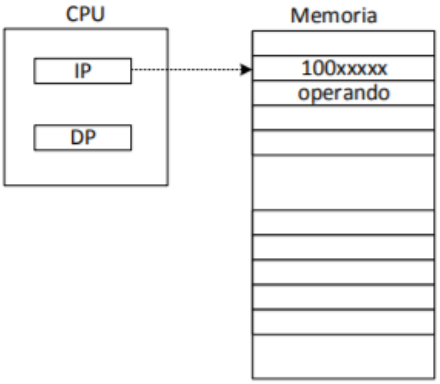



```
MOV (DP),AL     |01000000|
CMP (DP),AL     |01000001|
ADD (DP),AL     |01000010|
SUB (DP),AL     |01000011|
AND (DP),AL     |01000100|
OR  (DP),AL     |01000101|
MOV (DP),AH     |01000110|
CMP (DP),AH     |01000111|
ADD (DP),AH     |01001000|
SUB (DP),AH     |01001001|
AND (DP),AH     |01001010|
OR  (DP),AH     |01001011|
```

  - Formato F3 (011)
    - Categoria in cui rientrano le istruzioni dove l'operando destinatario si trova in memoria ed è indirizzato usando DP (solo le MOV)
    - Codifico su un unico byte l'istruzione. La fase di fetch consiste nel non fare niente: il contenuto da spostare è già presente nel processore, stessa cosa l'indirizzo da raggiungere.
    - La scrittura del destinatario avviene in fase di esecuzione.

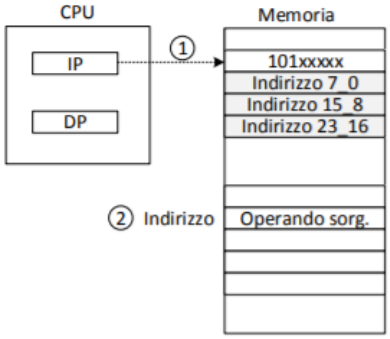
```
MOV AL,(DP) |01100000|
MOV AH,(DP) |01100001|
```

  - Formato F4 (100)
    - Categoria in cui rientrano le istruzioni dove l'operando sorgente è indirizzato in modo immediato, e sta su 8 bit.
    - L'istruzione è lunga due byte: il primo contiene l'istruzione, il secondo l'operando indirizzato in modo immediato. La fase di fetch consiste nel fare due letture consecutive.



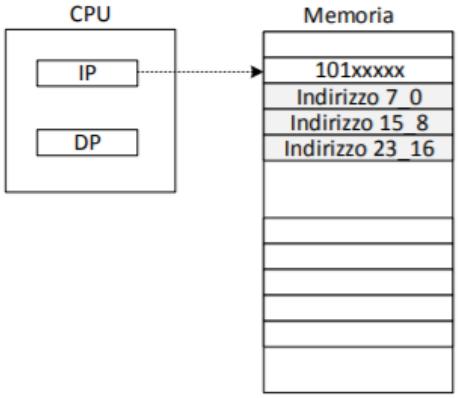

```
MOV $operando,AL |10000000|operando|
CMP $operando,AL |10000001|operando|
ADD $operando,AL |10000010|operando|
SUB $operando,AL |10000011|operando|
AND $operando,AL |10000100|operando|
OR  $operando,AL |10000101|operando|
MOV $operando,AH |10000110|operando|
CMP $operando,AH |10000111|operando|
ADD $operando,AH |10001000|operando|
[…]
```

- Formato F5 (101)
  - Categoria in cui rientrano le istruzioni dove l'operando sorgente è indirizzato in modo diretto. Ciò pongo direttamente l'indirizzo del sorgente.
  - In fase di Fetch l'operando sorgente deve essere riportato nel processore.
  - L'operazione sarà lunga 4 byte: uno di opcode e tre di indirizzo di memoria (24 bit per poter rappresentare qualunque indirizzo). Seguono tre cicli di lettura consecutivi a partire da IP. Ciò non basta: devo fare un'altra lettura all'indirizzo trovato: a quel punto ho raggiunto l'operando sorgente e posso porlo nel processore.



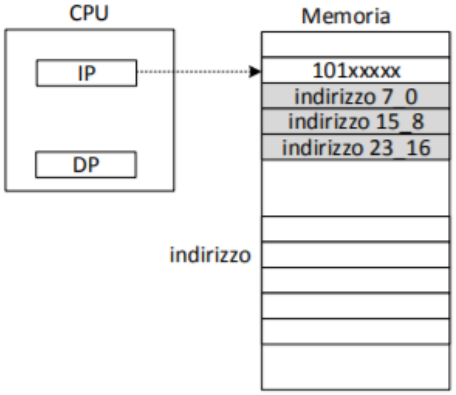

```
MOV indirizzo,AL |10100000| indirizzo |
CMP indirizzo,AL |10100001| indirizzo |
ADD indirizzo,AL |10100010| indirizzo |
SUB indirizzo,AL |10100011| indirizzo |
AND indirizzo,AL |10100100| indirizzo |
OR  indirizzo,AL |10100101| indirizzo |
MOV indirizzo,AH |10100110| indirizzo |
CMP indirizzo,AH |10100111| indirizzo |
[…]
```

- Formato F6 (110)
  - Categoria in cui rientrano le istruzioni dove l'operando destinatario è in memoria, indirizzato in modo diretto.
  - Il processore dovrà leggere 4 byte in memoria: uno per l'opcode, tre per l'indirizzo del destinatario.
  - La scrittura del destinatario avviene in fase di esecuzione.




```
MOV AL,indirizzo |11000000| indirizzo |
MOV AH,indirizzo |11000001| indirizzo |
```

- Formato F7 (111)
  - Uguale al precedente, raggruppa le istruzioni di controllo (CALL, JMP, Jcon) in cui ho un indirizzo di salto.
  - Utilizzo un byte per l'opcode, altri tre per l'indirizzo. In fetch abbiamo la lettura di 4 byte consecutivi, a partire da IP.

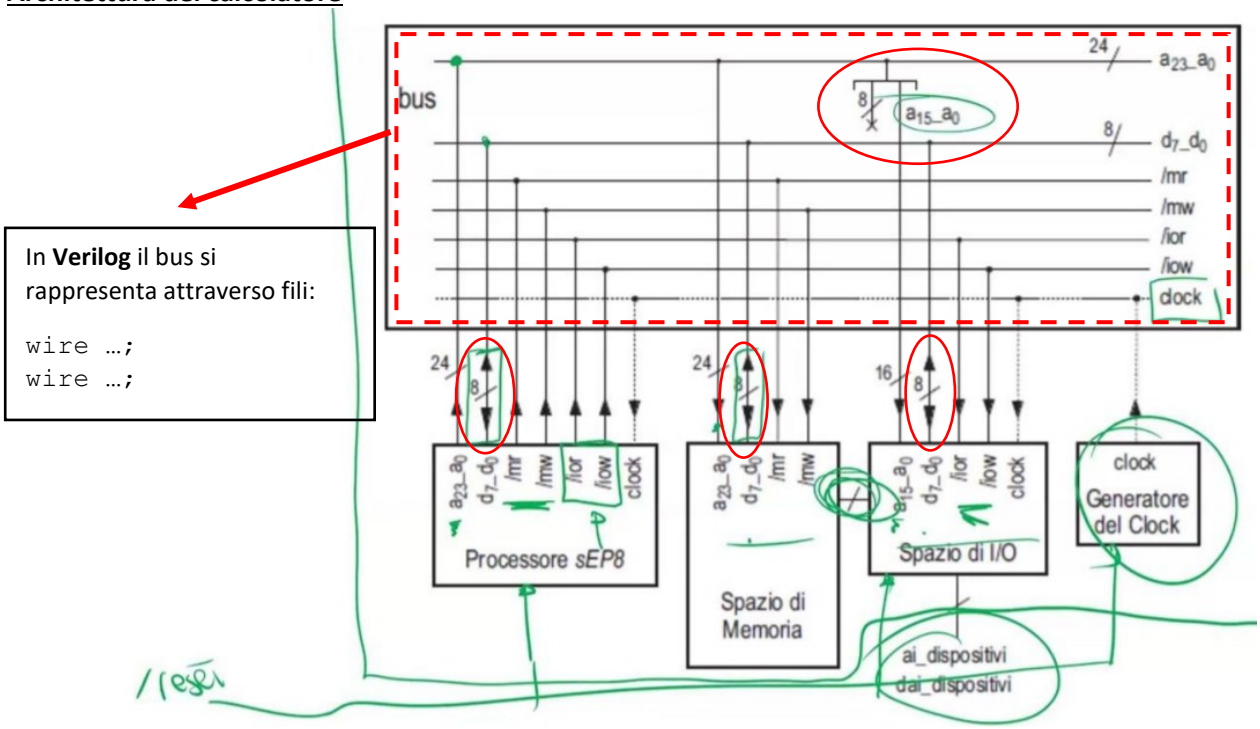



```
JMP  indirizzo  |11100000| indirizzo |
JE   indirizzo  |11100001| indirizzo |
JNE  indirizzo  |11100010| indirizzo |
JA   indirizzo  |11100011| indirizzo |
JAE  indirizzo  |11100100| indirizzo |
JB   indirizzo  |11100101| indirizzo |
[…]
CALL indirizzo  |11110011| indirizzo |
```

- Formato F1 (001) "formato delle varie ed eventuali"
  - Categoria in cui rientrano le istruzioni non classificabili nei formati precedenti: istruzioni I/O, MOV con uno dei registri a 24 bit.
  - Le azioni differiscono da un'istruzione a un'altra: si è preferito fare un unico formato dove ci si limita ad estrarre solo l'opcode. La gestione degli operandi viene rimandata alla fase di esecuzione.
  - La cosa è poco pulita, ma molto più semplice.

Lista completa delle istruzioni con formato corrispondente alle pagine 185-187 del libro di Corsini

Architettura del calcolatore



In **Verilog** il bus si rappresenta attraverso fili:

```
wire …;
wire …;
```

- Spogliamo il calcolatore e vediamo cosa è presente sulla rete di interconnessione

- Abbiamo:
  - **24 fili di indirizzo** che partono dal processore sEP8 e indicano l'indirizzo delle locazioni di memoria o delle porte di I/O dove vuole leggere e scrivere. Entrano in ingresso in tutti gli altri moduli. Attenzione allo spazio di I/O: avendo solo 64K mi bastano le 16 cifre meno significative per rappresentare tutti gli indirizzi possibili.
  - **Fili di dati**. Il processore legge e scrive singoli byte alla volta: ho bisogno di 8 fili che dovranno essere pilotati alternativamente dal processore e dagli altri dispositivi (porte tristate). Dobbiamo evitare cortocircuito sui fili di dati.
  - **Fili di controllo**. Attivi bassi con cui possiamo pilotare alcuni moduli a partire dal processore: /mr e /mw per leggere e scrivere in memoria, /ior e /iow per leggere e scrivere nello spazio di I/O. Chiaramente l'uso di queste variabili terrà conto delle leggi di temporizzazione viste per i cicli di lettura in memoria RAM.
  - **Generatore di clock**: tutti i moduli del calcolatore , tranne la memoria, sono collegati allo stesso generatore di clock.
  - **Fili di interconnessione tra interfacce e dispositivi**.
  - **Fili di comunicazione tra memoria video e adattatore grafico** (non vedremo)
  - Tenere conto anche della presenza dei **piedini per il reset** (il reset arriva contemporaneamente a tutti i moduli che ne hanno necessità). Per semplicità grafica la parte relativa al reset viene solitamente omessa.