Architettura degli elaboratori - lezione 14

Appunti di Davide Vella 2024/2025

Claudio Schifanella
claudio.schifanella@unito.it
Link al moodle :
https://informatica.i-learn.unito.it/course/view.php?id=3106

12/05/2025

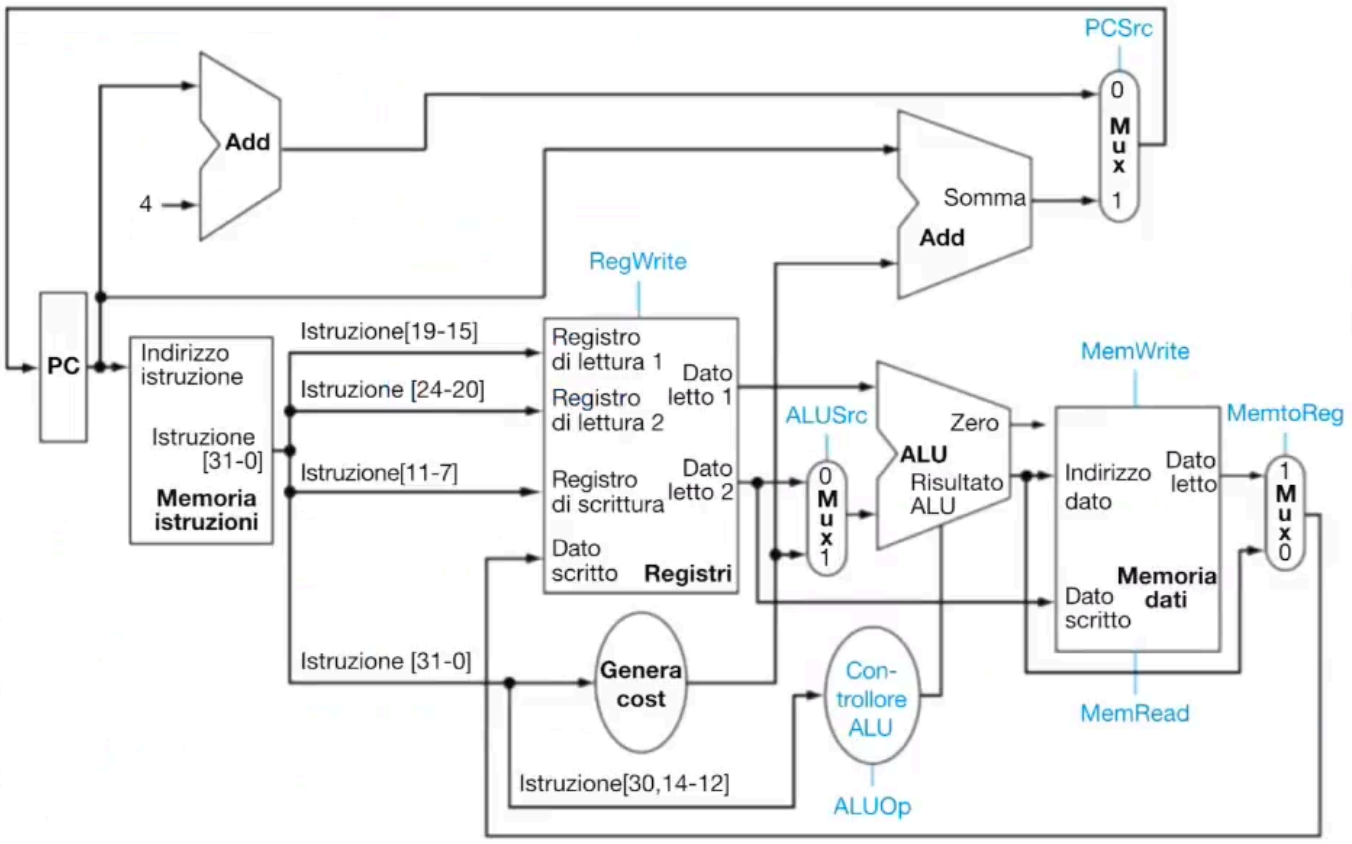
Contenuto

Recap formato delle istruzioni

Nome (posizione dei bit)	Campi					
	31:25	24:20	19:15	14:12	11:7	6:0
(a) Tipo R	funz7	rs2	rs1	funz3	rd	codop
(b) Tipo I	immediate[11:0]		rs1	funz3	rd	codop
(c) Tipo S	immed[11:5]	rs2	rs1	funz3	immed[4:0]	codop
(d) Tipo SB	immed[12,10:5]	rs2	rs1	funz3	immed[4:1,11]	codop

- **Tipo R**: i campi rs1 e rs2 contengono il numero dei registri sorgenti e rd contiene il numero del registro destinazione. L'operazione da eseguire è codificata nei campi funz3 e funz7
- **Tipo I, load**: rs1 è il registro base il cui contenuto viene sommato al campo immediato di 12 bit per ottenere l'indirizzo del dato in memoria. Il campo rd è il registro destinazione per il valore letto
- **Tipo S, store**: rs1 è il registro base il cui contenuto viene sommato al campo immediato di 12 bit (suddiviso in 2 gruppi) per ottenere l'indirizzo del dato in memoria. Il campo rs2 è il registro sorgente il cui valore viene copiato nella memoria
- **Tipo SB:** I registri rs1 e rs2 vengono confrontati. Il campo indirizzo immediato di 12 bit viene preso, il suo bit di segno esteso, fatto scorrere a sinistra di una posizione e sommato al PC per calcolare l'indirizzo di destinazione del salto

Recap segnali di controllo

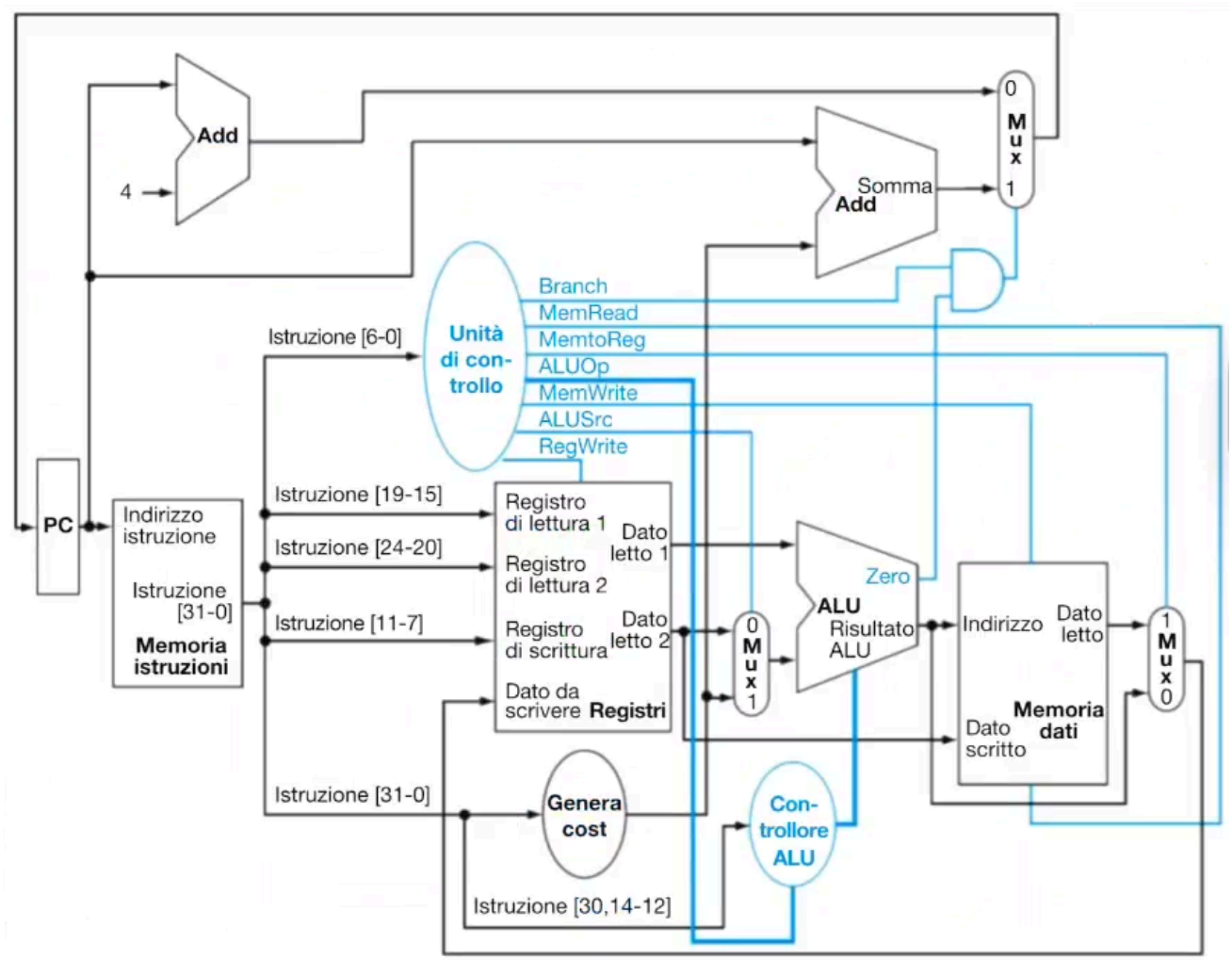


Dall'immagine sopra, ecco un recap dei segnali di controllo da 1 bit :

- RegWrite : se vale 1, prende il dato in ingresso da "Dato scritto" e lo scrive in "registro di scrittura" .
- MemRead : se vale 1, indica alla memoria se deve leggere un dato contenuto in "indirizzo dato"(serve perché l'indirizzo presentato in "indirizzo dato" potrebbe non essere valido,

quindi c'é bisogno di un controllo esplicito che mi dica che voglio leggere da quell'indirizzo).

- MemWrite : se vale 1, devo scrivere in memoria il dato che trovo in "dato scritto" nell'indirizzo contenuto in "indirizzo dato".
- MemtoReg : è il segnale di controllo di un multiplexer che decide se prendere il dato letto dalla memoria o restituito dalla ALU (ad esempio, quando facciamo una load, vogliamo 1, perché leggiamo un dato in memoria. Invece, se facciamo una add, vogliamo 0, perché facciamo la somma tra due valori e mettiamo poi il risultato in un registro)
- PCSrc : è il segnale di controllo di un multiplexer che vale 0 se il PC deve essere incrementato semplicemente di 4 oppure 1 se il PC deve essere incrementato di un offset generato da "genera cost" (ovvero quando facciamo un salto). (NOTA, questa linea è collegata in AND alla linea di uscita della ALU "zero" e all'uscita dell'unità di controllo "branch", vedi immagine sotto*).
- ALUSrc : è il segnale di controllo di un multiplexer che pilota l'ingresso 2 della ALU. Vale 0 se dobbiamo prendere un dato da un registro (quindi dal register file) oppure 1 se prendiamo un dato estrapolato dall'immediato dell'istruzione (e quindi generato da "genera cost").



-

> **Note**

Come fa l'unità di controllo a sapere che un'istruzione è di branch/load/store...?
Guardando il codop.

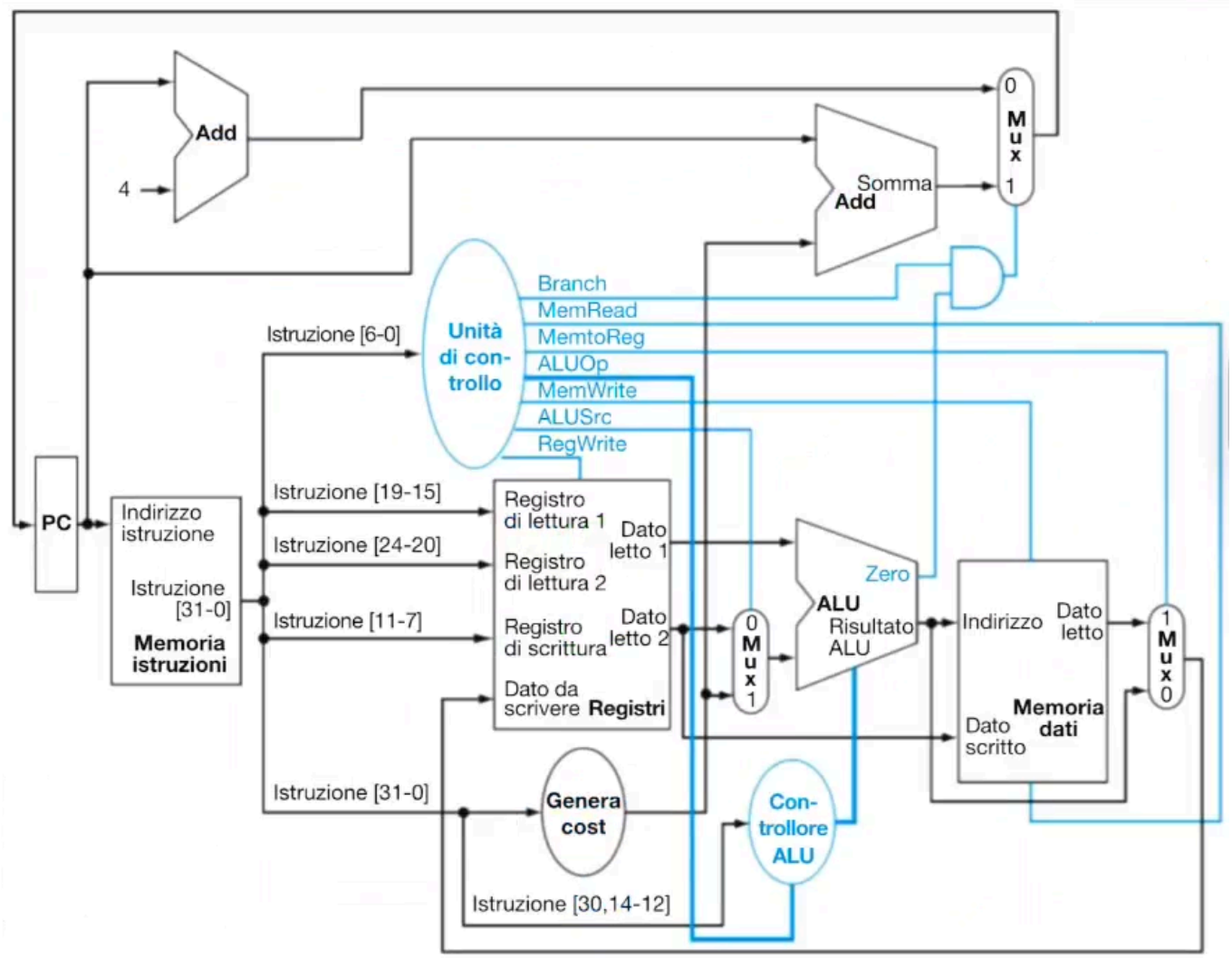
Valore delle linee di controllo per ogni caso

Di seguito un'immagine per ricapitolare il valore di ogni linea di controllo per ogni istruzione (vista ora) :

Istruzione	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
Tipo R	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

Esempio (utile per l'esame)

Durante l'esame, può essere che capiti un esercizio del genere, ovvero, data un'istruzione e l'immagine del circuito, venga chiesto cosa succede :



Istruzione :

```
add x1, x2, x3
```

Partiamo con dei ragionamenti ovvi che possiamo fare, ovvero ciò che non viene usato :

- Genera cost : non viene sicuramente usato perchè "add" è un'istruzione di tipo R e quindi non contiene immediati
- La ALU "add" che somma un immediato al PC : non viene usato perché l'istruzione "add" non è un salto
- Memoria dati : non viene usata perché "add" non accede alla "memoria dati" (ram) perché i valori cercati (x2, x3) sono contenuti nei registri (cache).
  Poi possiamo procedere :

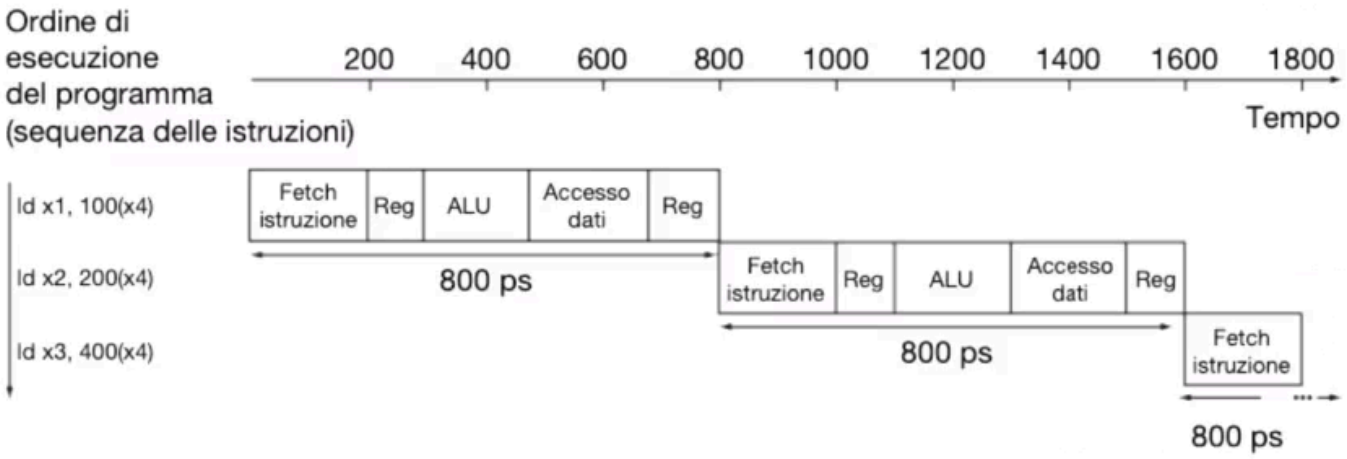
1. Il PC espone il proprio contenuto alla memoria delle istruzioni in "indirizzo di istruzione" e incomincia la fase di "fetch".
2. L'istruzione esce dalla memoria delle istruzione e si dirama in vari punti (registro di lettura 1 (x2, ovvero "00010") registro di lettura 2 (x3, ovvero "00011"), registro di scrittura (x1,

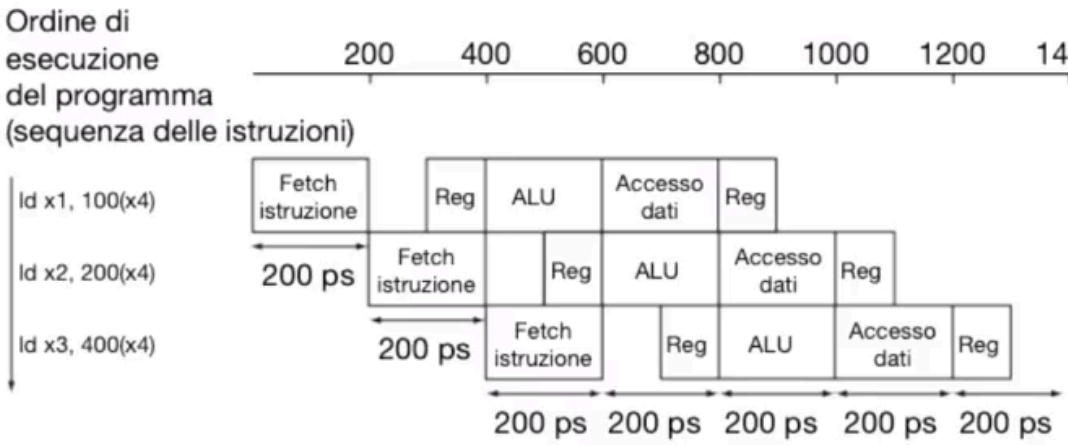
"00001"), unità di controllo, controllore ALU e genera cost (anche se non viene usata, è pur sempre collegata)).

3. L'unità di controllo ora sta facendo "decode" dell'istruzione e sta mettendo i valori adeguati a tutte le linee di controllo (es : branch = 0, MemRead = 0, MemtoReg = 0 (noi vogliamo che al register file torni il valore calcolato dall'ALU, non dalla memoria dati), ALUOp = (qualcosa), MemWrite = 0, ALUSrc = 0 (vogliamo dato letto 2), RegWrite = 1)
4. Il register file presenta alla ALU da "dato letto 1" e "dato letto 2" i due valori da sommare.
5. Se il controllore ALU ha presentato all'ALU i valori dei 4 bit, allora l'ALU fa la somma dei valori. Altrimenti, se i valori non sono ancora stabili, la ALU presenterà il valore della somma con i vecchi 4 bit fino a quando però, i 4 bit diventano stabili e la ALU fa il giusto calcolo.
6. Il valore calcolato dall'ALU va verso il multiplexer con ingresso "MemtoReg" che vale 0, quindi fa passare il valore e arriva al register file.
7. Siccome RegWrite vale 1, il valore in "dato da salvare", viene salvato nel registro preso da "registro di scrittura". ATTENZIONE : il valore viene salvato al prossimo fronte di salita, non prima.
   È finito questo esempio. Possono ovviamente essere richieste cose diverse (ad esempio, una load, una store, un salto...).

Pipeline (approfondimento non richiesto)

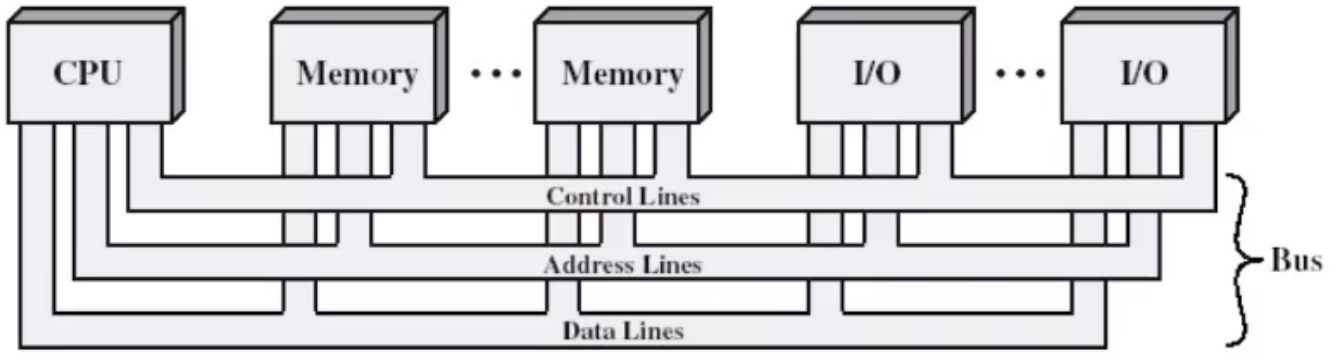
In maniera semplice, il pipelining è l'ottimizzazione dell'esecuzione del programma facendo in modo parallelo più istruzioni. Di seguito un immagine per chiarire un po' il concetto :





I BUS

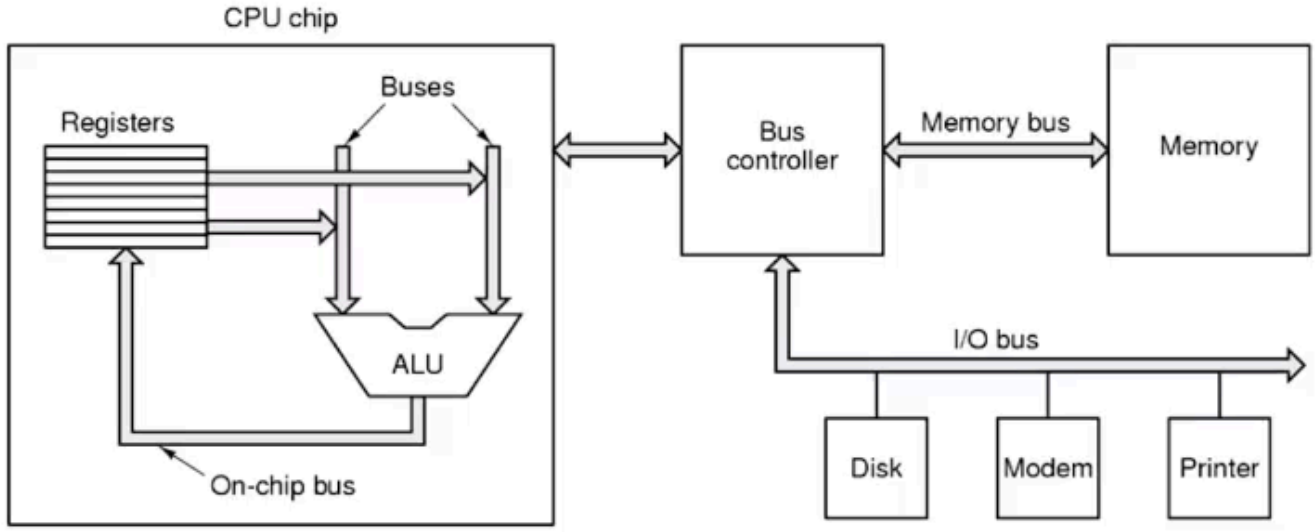
I bus sono dei canali di trasporto che portano delle informazioni da un componente all'altro (non da una memoria all'altra, sono cose diverse).
Ci sono molteplici bus. Variano sia in architettura, in sistemi moderni o tradizionali... Noi vedremo un'astrazione di bus, non vedremo una cosa molto moderna.
Di seguito un esempio di utilizzo di bus in una architettura di Von Neumann :



> **Note**
>
> Di cavi bus ne esistono tantissimo. Molte cose possono essere definite bus. Ad esempio troviamo : bus interni al processore, bus esterni che collegano vari componenti (PCIe),

bus esterni al calcolatore (cavi USB, cavi ethernet)... Banalmente, anche il WIFI, e quindi l'aria, può essere visto come un bus di comunicazione

Un altro esempio di quanto detto sopra nella nota è questa immagine che chiarisce meglio la struttura :



Tipi di bus

I bus possono essere linee di :

- Dati : il numero di linee (larghezza del data bus) determina il numero di bit che possono essere trasmessi alla volta, ha un un impatto sulle prestazioni del sistema.
- Indirizzo : permettono di individuare la sorgente/destinazione dei dati trasmessi sul data bus.
- Controllo : controllano l'accesso e l'utilizzo delle linee di dati e di indirizzo.

Dispositivi attivi e passivi

I dispositivi collegati ad un bus si dividono in :

- Attivi, possono decidere di iniziare un trasferimento, in genere sono collegati al bus per mezzo di un particolare chip detto bus driver
- Passivi, rimangono in attesa di richieste, in genere sono collegati per mezzo di un chip detto bus receiver
  I dispositivi che si comportano sia come attivo che come passivo (es la CPU) sono collegati attraverso un chip combinato, il bus transceiver

Progettazione dei bus

I principali problemi nella progettazione di un bus riguardano :

- Larghezza del bus (numero di linee)
- Arbitraggio, come scegliere tra due dispositivi che vogliono diventare contemporaneamente arbitri dello stesso bus
- Funzionamento del bus, come avviene il trasferimento dei bit

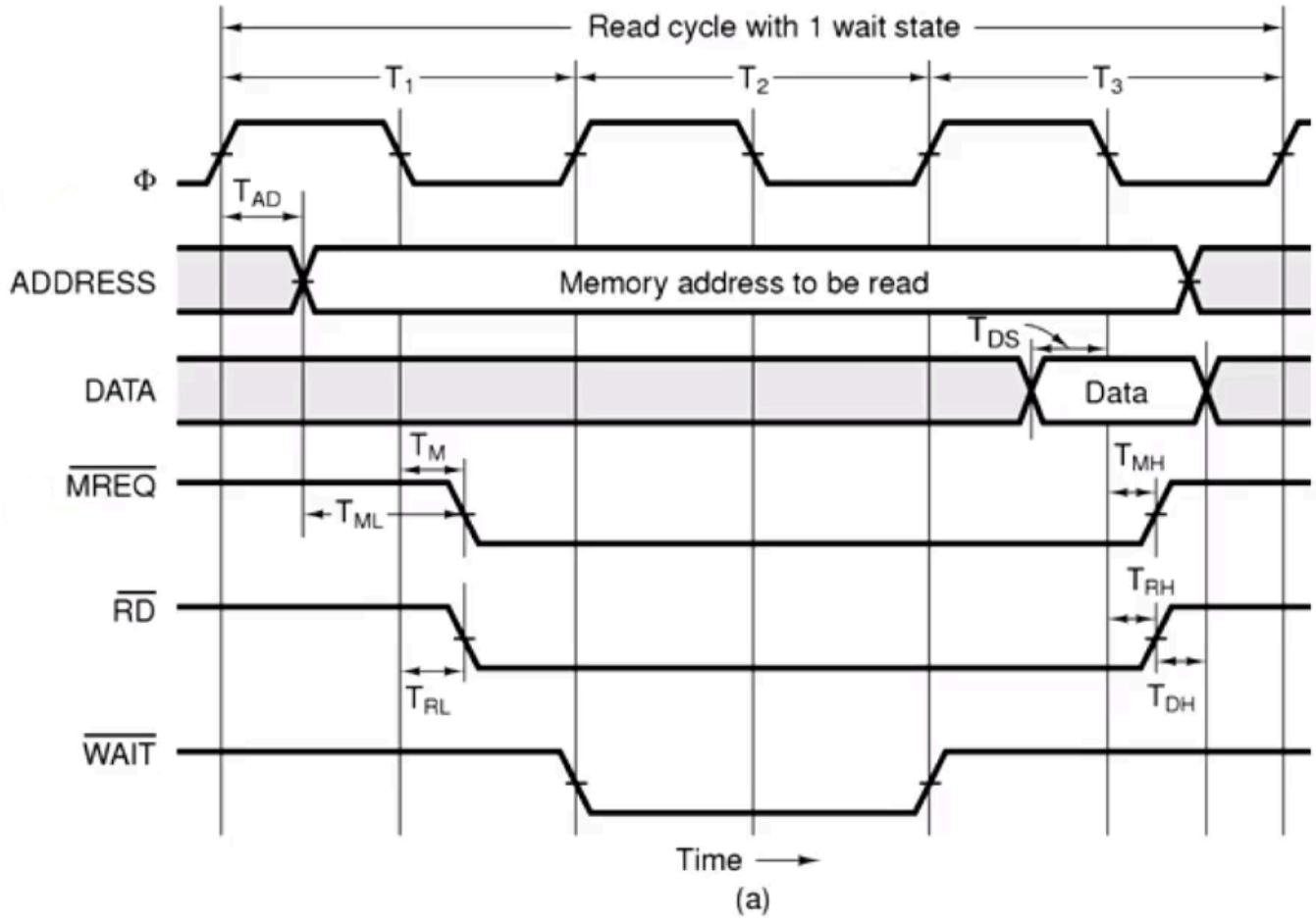
Categorie dei bus

I bus si possono dividere in due categorie :

- Bus sincroni, usano una linea di clock comune (pilotata da un oscillatore), con tutte le attività che avvengono in multipli interi del ciclo di clock. Esempi classici includono ISA, PCI e DDR. Le frequenze possono variare da pochi MHz (ISA) a diversi GHz (DDR5, PCIe).
- Bus asincroni, non usano un clock globale. Ogni transazione può richiedere tempi diversi, sincronizzandosi tramite segnali di handshaking.

Bus sincrono

Vediamo un esempio di bus sincrono :



Prendiamo il caso in cui la CPU (master) deve leggere qualcosa dalla memoria (slave).
Chiariamo delle cose (alcune convenzioni):

- I fronti del clock sono obliqui per rispecchiare un po' di più la realtà, ma a noi questa cosa non interessa.
- Possiamo notare chiaramente la differenza tra alcune fasce.
  - Fasce "ampie" : queste fasce (come address e data) sono formate da più bit. In queste fasce, quando il segnale è colorato di grigio, vuol dire che il segnale non è stabile in quel tratto, non c'é il valore che mi serve in quel momento. Quando è bianco, vuol dire che il valore è stabile.
  - Fasce "stette" : queste fasce (come MREQ, RD, WAIT) sono formate da solo 1 bit. Queste fasce possono presentare alcune righe sopra. Quando una riga è presente sopra un segnale, vuol dire che viene asserito quando il valore vale 0 e non è asserito quando il valore vale 1. Ad esempio, prendiamo RD. Quando RD (read) vale 1, noi NON vogliamo leggere, mentre quando vale 0, noi vogliamo leggere.
  Specificato ciò, vediamo cosa fa il processore per leggere un valore dalla memoria :

1. Il processore mette l'indirizzo sul bus degli indirizzi.
2. Il processore mette a 0 (asserisce) il valore di MREQ (memory request) e RD (read) per dire che vuole accedere alla memoria e che vuole leggere.
3. La memoria si accorge che MREQ e RD sono stati asseriti (la memoria non presta caso al cambiamento dell'indirizzo nel bus apposito, potrebbero esserci diverse cose che fanno cambiare quell'indirizzo, non per forza un accesso alla memoria).
4. Adesso la memoria asserisce il valore di WAIT, per dire al processore che deve aspettare il dato stabile.
5. La memoria legge il dato richiesta all'indirizzo richiesto e lo carica sul bus "DATA" e mette WAIT ad 1 (per dire che è disponibile il dato).
6. Il processore a questo punto legge il dato e mette ad 1 sia MREQ sia RD (per dire alla memoria che non gli serve più).
   Il processo è finito, ma c'é una cosa da specificare. Possiamo notare che il segnale WAIT è messo ad 1, ma in realtà in quel momento il segnale nel bus DATA non è ancora stabile, perché? Perché il processore scrive il dato richiesto in un registro, ovvero in dei flip flop, i quali scrivono solo su un fronte. Allora, la memoria notifica al processore in un punto, ma poi la memoria è sicura che il dato sarà disponibile sul prossimo fronte (in questo caso un fronte di discesa).
   Il segnale di WAIT non è essenziale (per come abbiamo strutturato noi ora questa parte, sì), ma se noi conosciamo le specifiche della memoria e sappiamo, ad esempio, che la memoria legge in 3 cicli di clock, il segnale non serve.