Lezione 40 pipeline

lunedì 11 marzo 2024 11:14

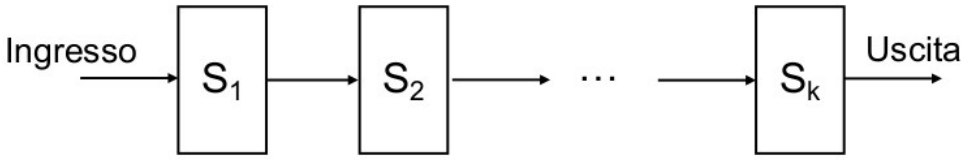
Cerchiamo di rendere l'esecuzione i nostri programmi : il multi ciclo visto finora non va male in quanto c'è anche il riuso dei componenti per svolgere più istruzioni diverse (spezzettamento istruzioni in micro istruzioni). Ma il problema di questa architettura è che è sequenziale in quanto esecuzione di un'istruzione si sviluppa secondo **fetch / decode / execute**. Quindi in generale con questo tipo di processore ci potrebbe un sottoutilizzo delle risorse , con singola esecuzione di istruzione nel ciclo. Andiamo a stimare questo tempo :

$$T_{esec} = n_{cicli} \cdot T_{clock} \text{ equivalente a } T_{esec} = \frac{n_{cicli}}{f_{clock}}$$

Dove ncicli (numero di cicli) si calcola cosi':

$$n_{cicli} = n_{insn} \cdot CPI$$

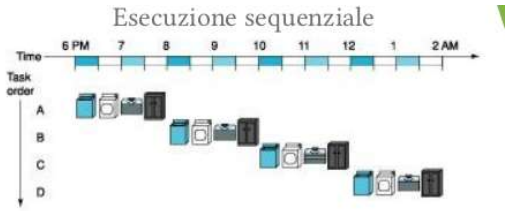
Dove CPI sta per clock process istruction (numero colpi clock necessari per eseguire istruzione). Vediamo ora come poter "aggiustare" / velocizzare queste equazioni : per esempio possiamo aumentare la frequenza di clock (legge di Moore) , ridurre il numero di istruzioni (differenza tra cisc e risc) . Andiamo ora a vedere l'architettura **PIPELINE :** architettura all'interno del quale vi è un "tubo" , all'interno del quale scorrono successivamente varie istruzioni : **il processore esegue contemporaneamente varie parti delle istruzioni che sono nel tubo.** L'esecuzione di un'istruzione viene suddivisa in passi o stati : ogni istuzione li deve attraversare tutti. In dettaglio :



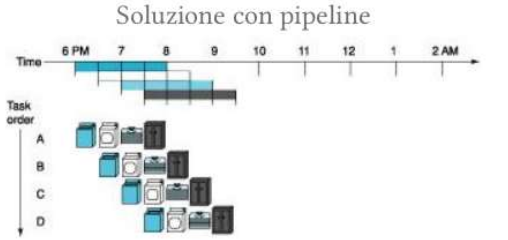
Vediamo un esempio :

Attività elementari per fare il bucato:

- Lavaggio
- Asciugatura
- Stiratura
- Riordino



Nella soluzione con pipeline viene avviato il ciclo di lavaggio successivo mentre quello precedente è ancora in esecuzione in un'altra fase



Quindi in dettaglio :

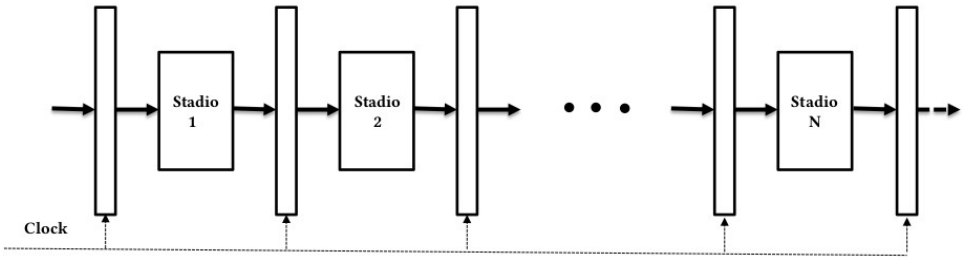
Class	Instruction	Syntax	Semantics
HW	Non-operational	`nop`	No operation (beyond the fetch phase)
L/A	Sum	`addq %regsorg, %regdest`	regdest = regsorg + regdest
	Subtraction	`subq %regsorg, %regdest`	regdest = regdest - regsorg
	Logic product	`andq %regsorg, %regdest`	regdest = regsorg and regdest
	Logic sum	`orq %regsorg, %regdest`	regdest = regsorg or regdest
	Logic negation	`notq %register`	register = not register
L/S	Loading of word	`movq offset(%regbase), %regdest`	regdest = memory[offset+regbase]
	Storage of word	`movq %regsorg, offset(%regbase)`	memory[offset+regbase] = regsorg
J	jump if flag X == 1	`jX displacement`	if flag X == 1 then RIP = RIP + displacement

- Il formato istruzione è identico a quello del processore multiciclo (ma supportiamo solo istruzioni a 64 bit)

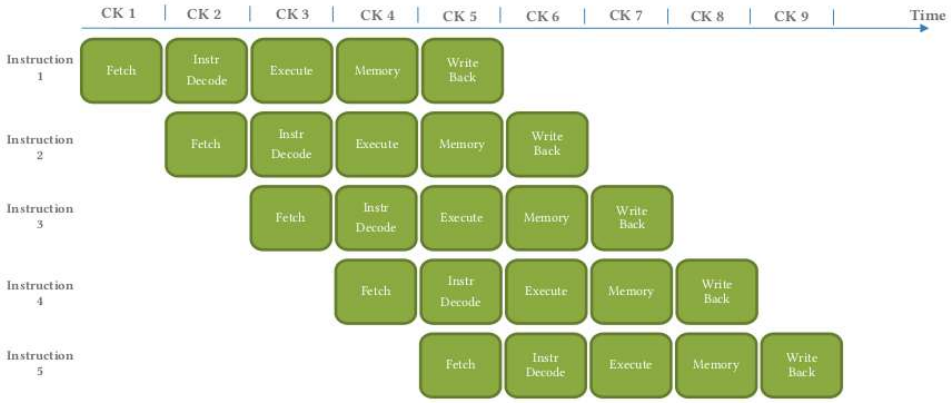
Per studiare una qualunque architettura pipeline , andiamo a vedere i passi elementari:

1. Nop :
    a. Eseguo fetch, recupero dalla memoria , incremento rip , eseguo decode
    b. Non devo fare niente per fare la NOP
2. Istruzioni logico aritmetiche
    a. Prelevo il dato (fetch)
    b. Eseguo fase decode : vedo gli operandi dell'istruzione logico-aritmetica
    c. Possibili operandi sono solo i registri
    d. Nella fase di exec : eseguo effettivamente l'operazione (ALU fa cose)
    e. Dobbiamo scrivere il risultato nel registro (Write back)
3. Load: registro base + offset
    a. Prelevo il dato (fetch)
    b. Leggo registro base (trattato come registro sorgente) decode
    c. Calcolo indirizzo da dove prelevare il dato (execute) -> offset+base
    d. Leggo il dato della locazione di memoria puntato da offset+base -> memory
    e. Scrivo il contenuto nel registro di destinazione -> write back
4. Store: scrittura in memoria ; non ho write back
    a. Prelevo istruzione (fetch)
    b. Leggo registro base (trattato come registro sorgente) decode
    c. Calcolo indirizzo da dove prelevare il dato (execute) -> offset+base
    d. Leggo il dato della locazione di memoria puntato da offset+base -> memory
5. Jmp : salto condizionale
    a. Prelevo istruzione (fetch)
    b. leggo registro base (trattato come registro sorgente) decode
    c. Calcolo indirizzo da cui prelevare il dato , ricordando che il bit di flag è impostato a 1 -> execute
    d. Aggiorno PC (program counter) in base al valore del flag

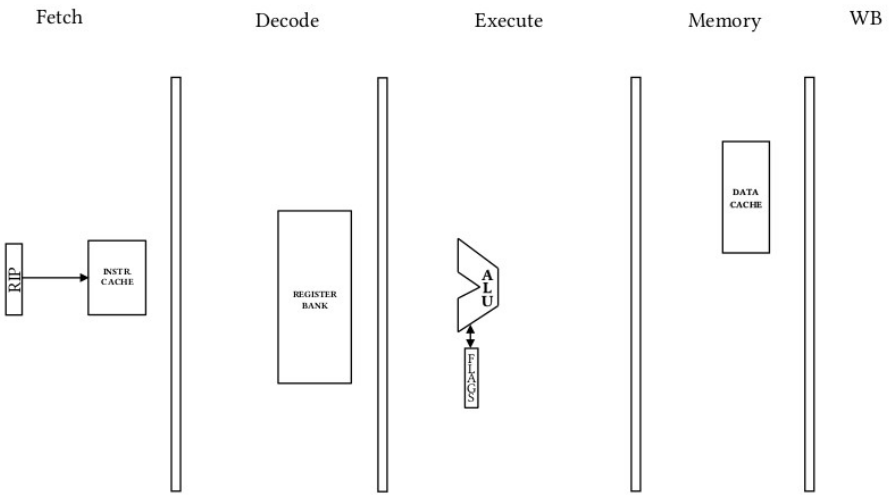
In generale si hanno da 2 a 5 fasi .**Ma in questo caso per forza 5 fasi! Quindi il vantaggio è critical path più breve . Inoltre disaccoppio il mio circuito in 5 sotto circuiti -> aumento frequenza clock -> diminuisce tempo esecuzione.** Andiamo a vedere lo schema base :



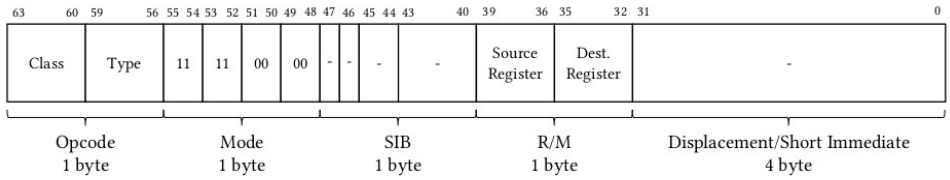
Ogni fase è un circuito a se stante , quindi vengono disaccoppiati da dei registri tampone. In dettaglio :

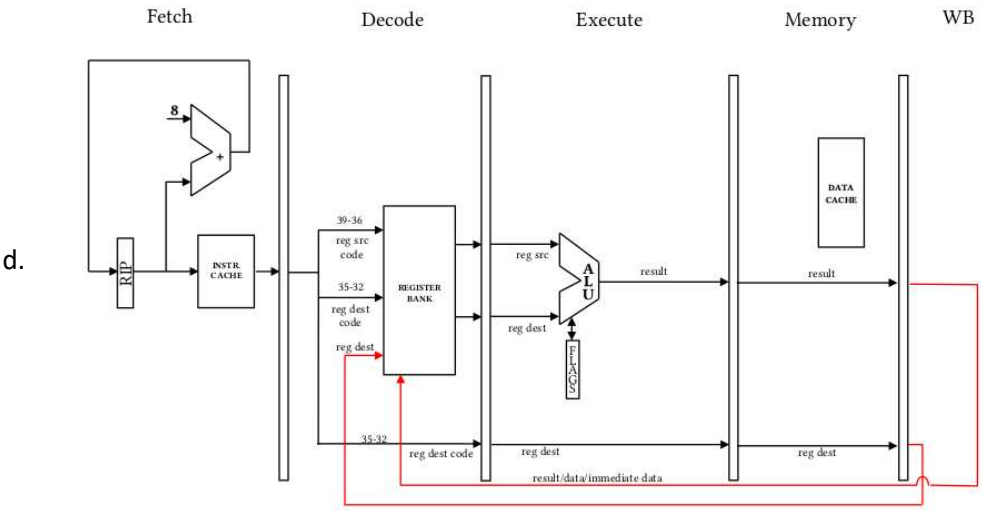


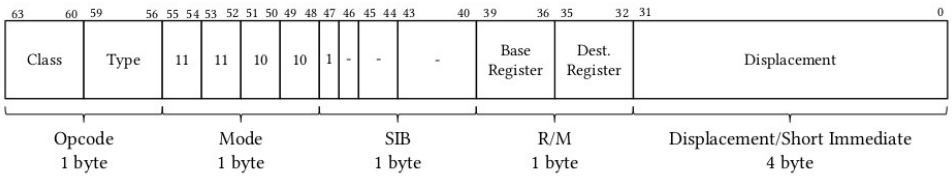
In generale lo schema per ogni istruzione è il seguente :

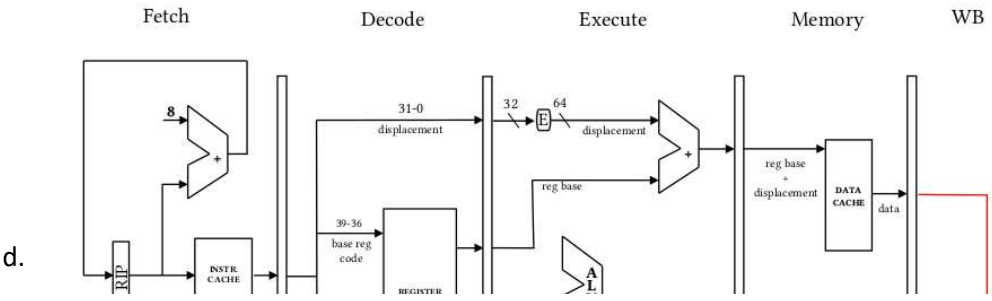


Andiamo a vedere effettivamente come viene implementato il processore per ogni tipo di istruzione :

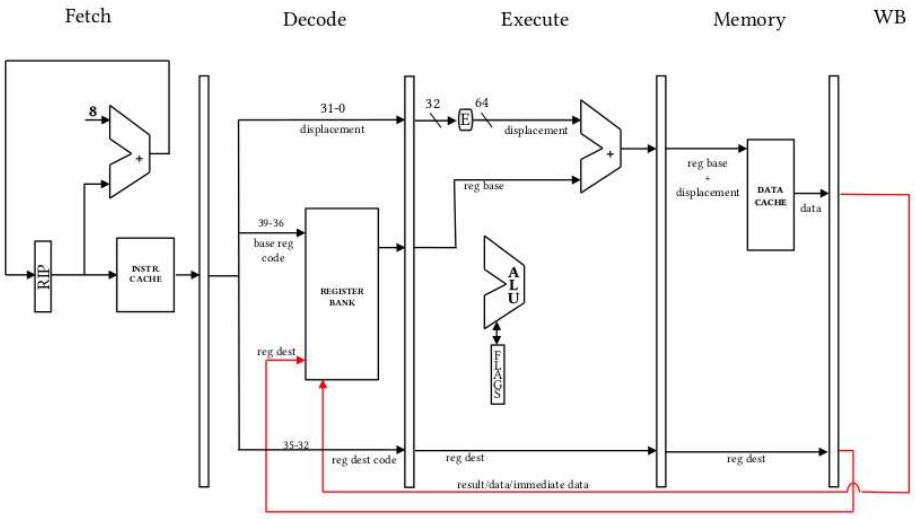
1. Logico/aritmetico
   a. Vincolo : operandi soltanto a 64 bit , non spiazzamento e dati immediati , solo registri e spiazzamento massimo a 64 bit
   b.


   c. Lo schema completo è il seguente :
   d.


2. Istruzione di load :
   a. Esiste registro base , non uso costanti, non memoria , operandi a 64
   b.


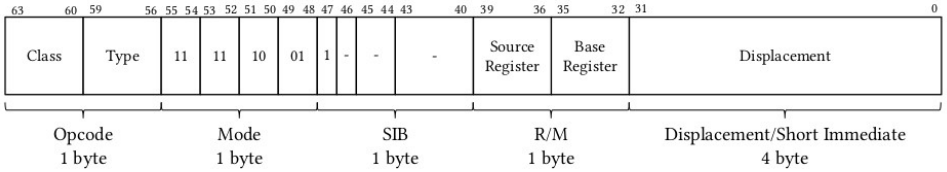
   c. Lo schema completo è il seguente:
   d.


d.

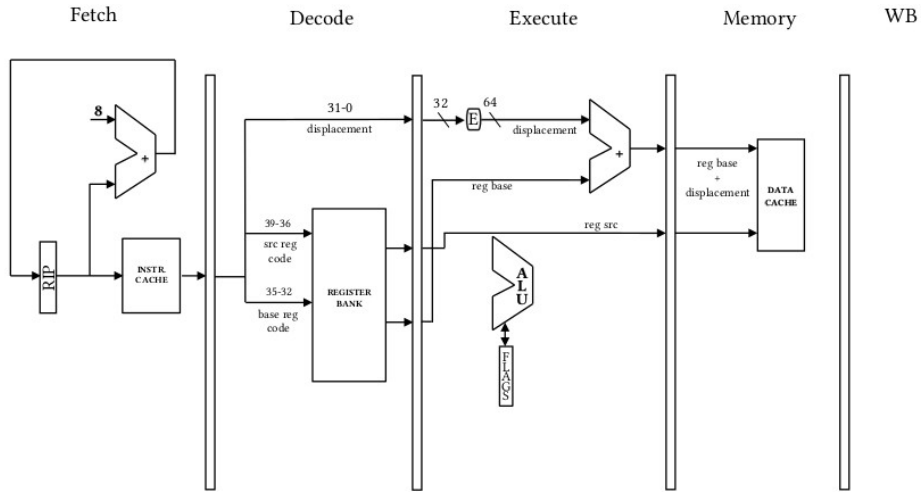


e. Notiamo che la "E" sta per estensione segno : porta da 32 a 64 bit

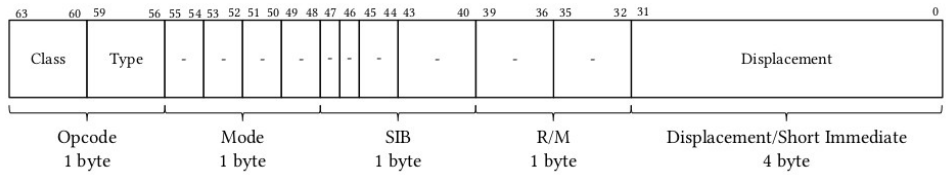
3. Store :
   a. Sorgente registro , dst registro
   b.



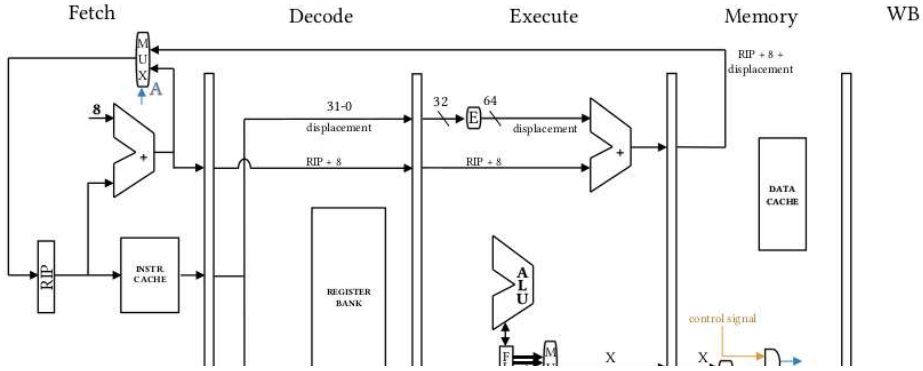
   c. Il circuito è il seguente:



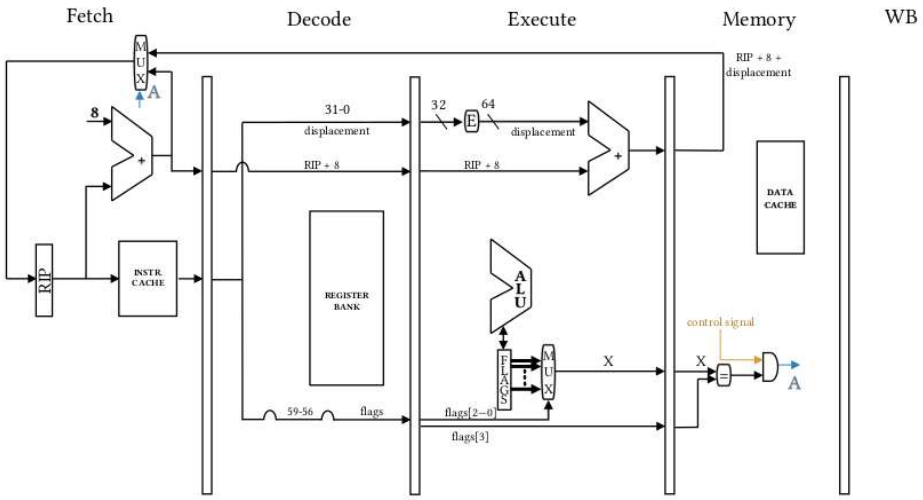
   e. Il write back è sprecata : ci passo ma non faccio nulla
4. Salto condizionale :
   a. Spiazzamento , bit del registro flag se salto deve essere fatto o meno
   b.



   c. Il circuito è il seguente:



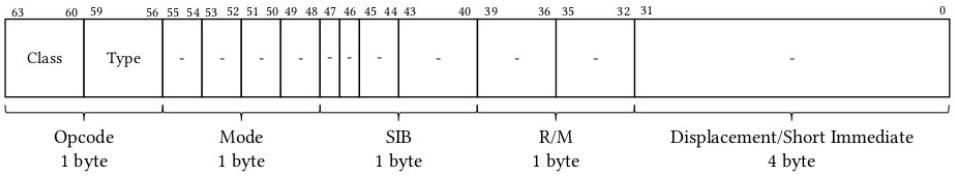
d.



e. Il salto viene scoperto solo nello stato di execute (2 stati dopo) rispetto a Rip->Rip+8

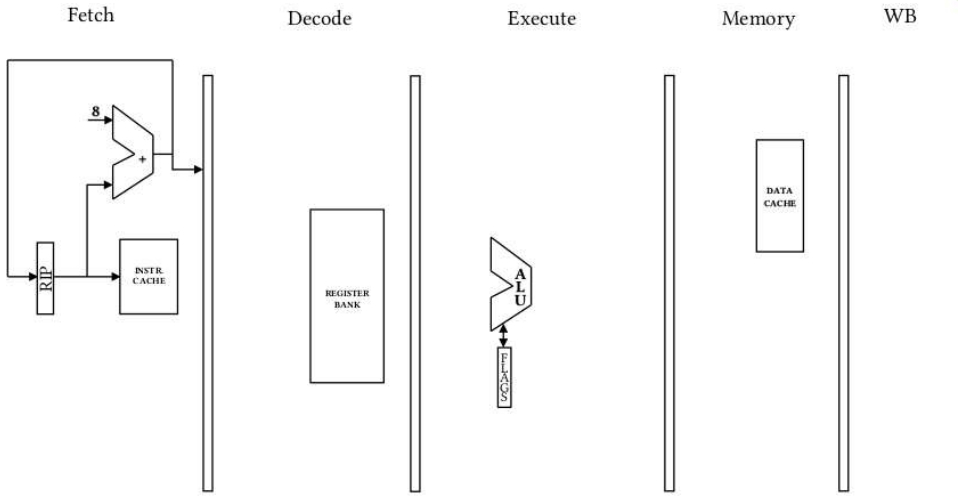
5. Nop
   a. 2 fasi

b.



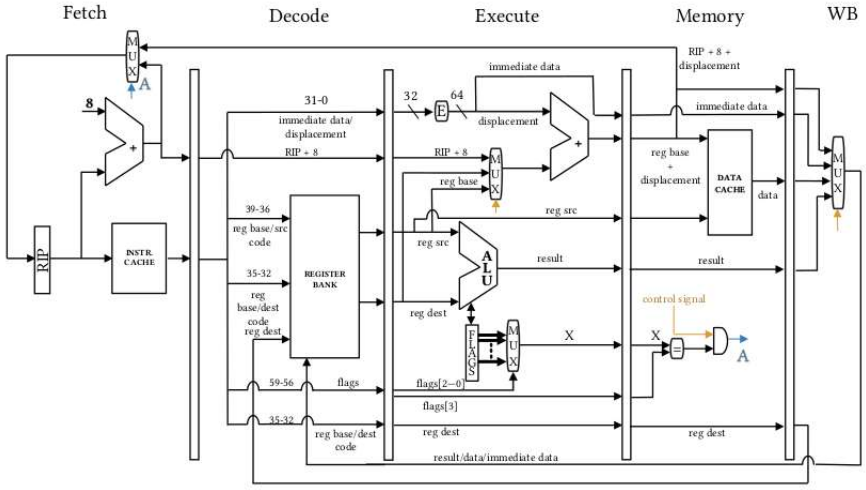
c. Quindi il circuito è il seguente :

d.



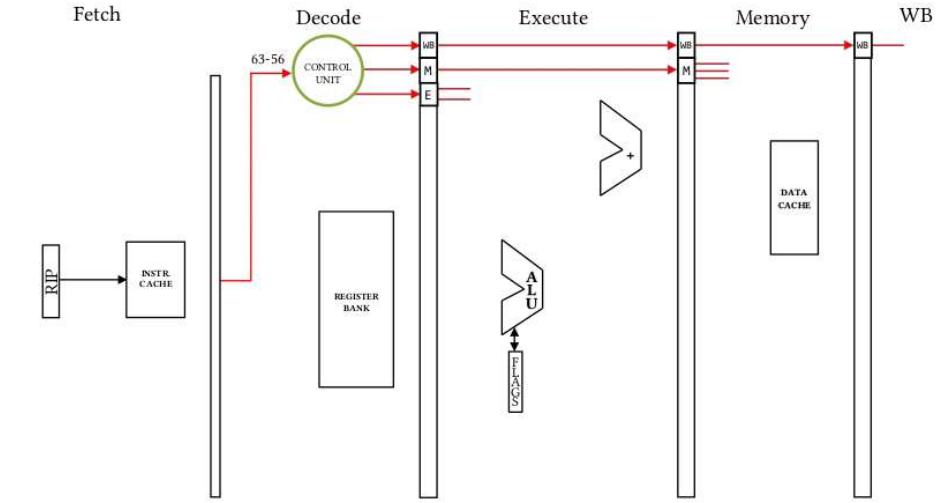
e. <u>Notiamo che sprechiamo le altre fasi !</u>

Andiamo ora a vedere il datapath completo : da notare che utilizziamo dei mux per selezionare il valore corretto (da propagare) :



Notiamo che per ogni stadio ci sono 3 segnali di controllo. Quanto detto finora però non tiene conto dell'unità di controllo , la quale serve per vedere come si propagano i segnali tra le varie

tubazioni (registri di interfaccia) : vediamolo come circuito combinatorio che calcola delle funzioni booleane:



Quindi lo schema finale è il seguente: