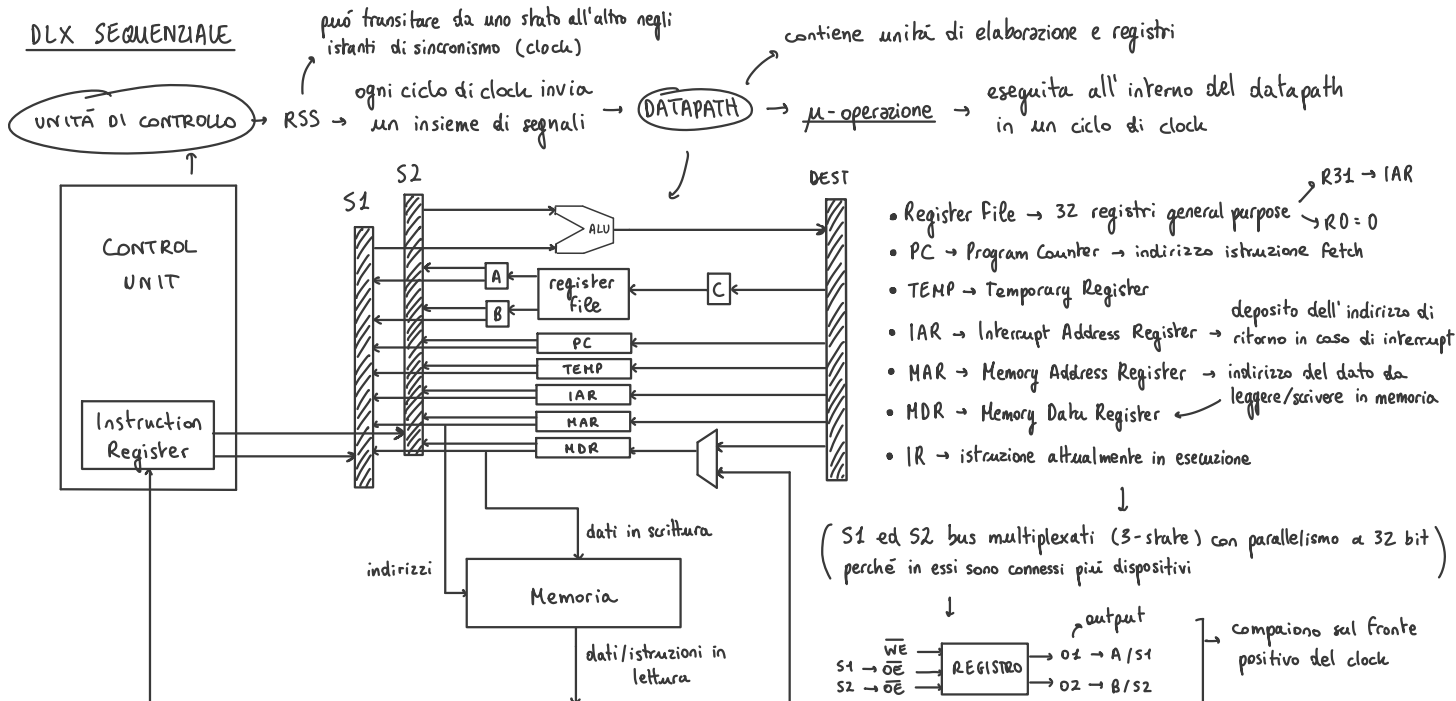


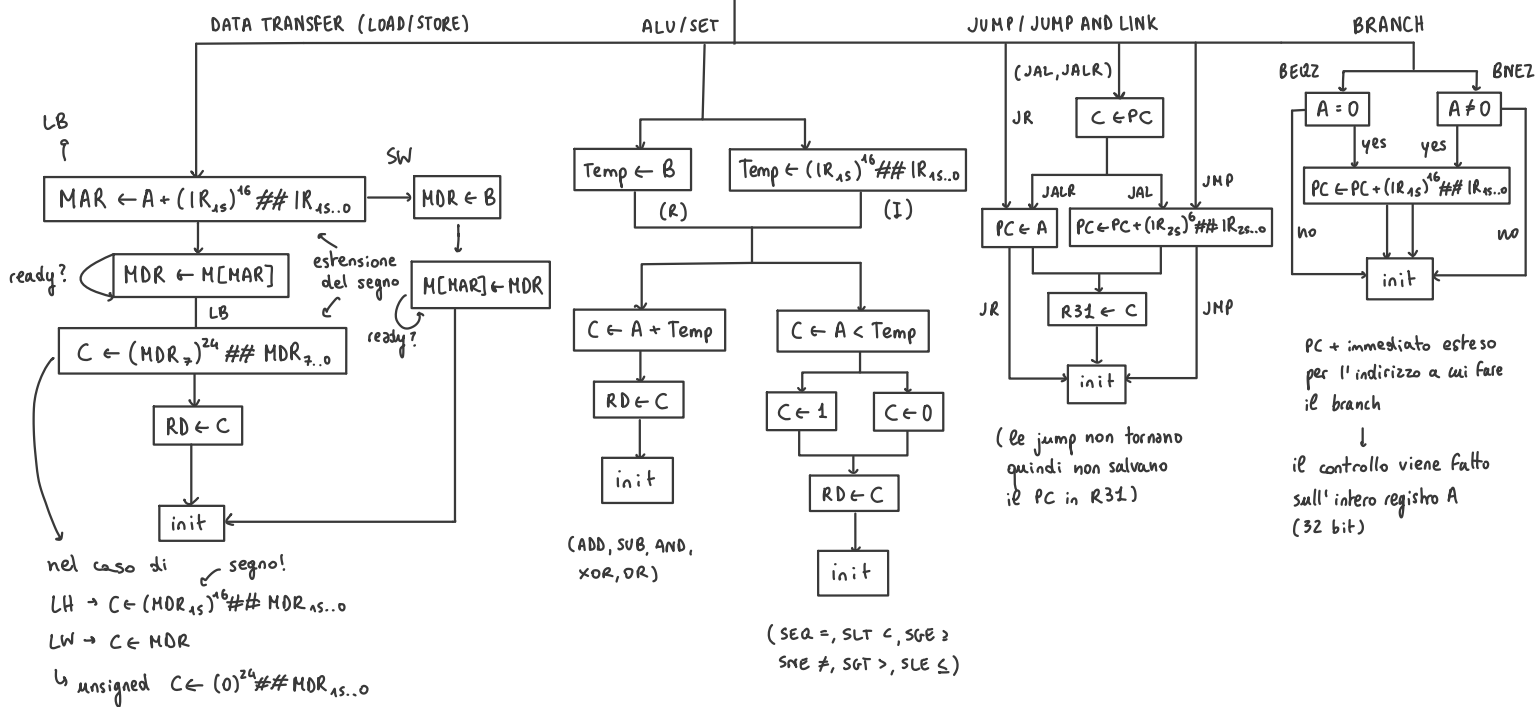
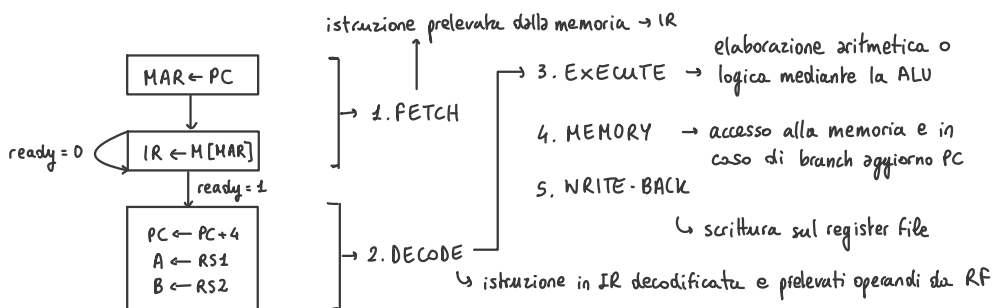
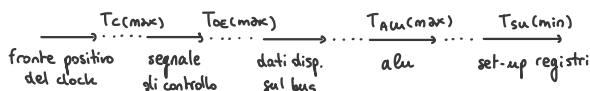
DLX SEQUENZIALE



per passare dal bus sorgente al bus destinazione si deve necessariamente passare per l'ALU; qualsiasi dato che viene scritto in memoria passa da MDR

massima frequenza di funzionamento del datapath → $T_{cu} > T_{c(max)} + T_{oe(max)} + T_{au(max)} + T_{su(min)}$

$$f_{cu(max)} = \frac{1}{T_{cu}}$$

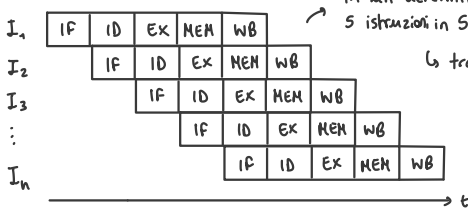


clock × instruction

$$CPI = \sum_{i=1}^n (CPI_i \cdot \frac{N_i}{N_{tot \text{ istruzioni}}})$$

DLX PIPELINED → più istruzioni in esecuzione contemporaneamente nella sequenza definita inizialmente → CPI = 1

T_p → pipeline cycle → a regime = clock cycle → idealmente CPI = 1



in un determinato istante 5 istruzioni in 5 fasi diverse → ad ogni ciclo di clock un'istruzione termina

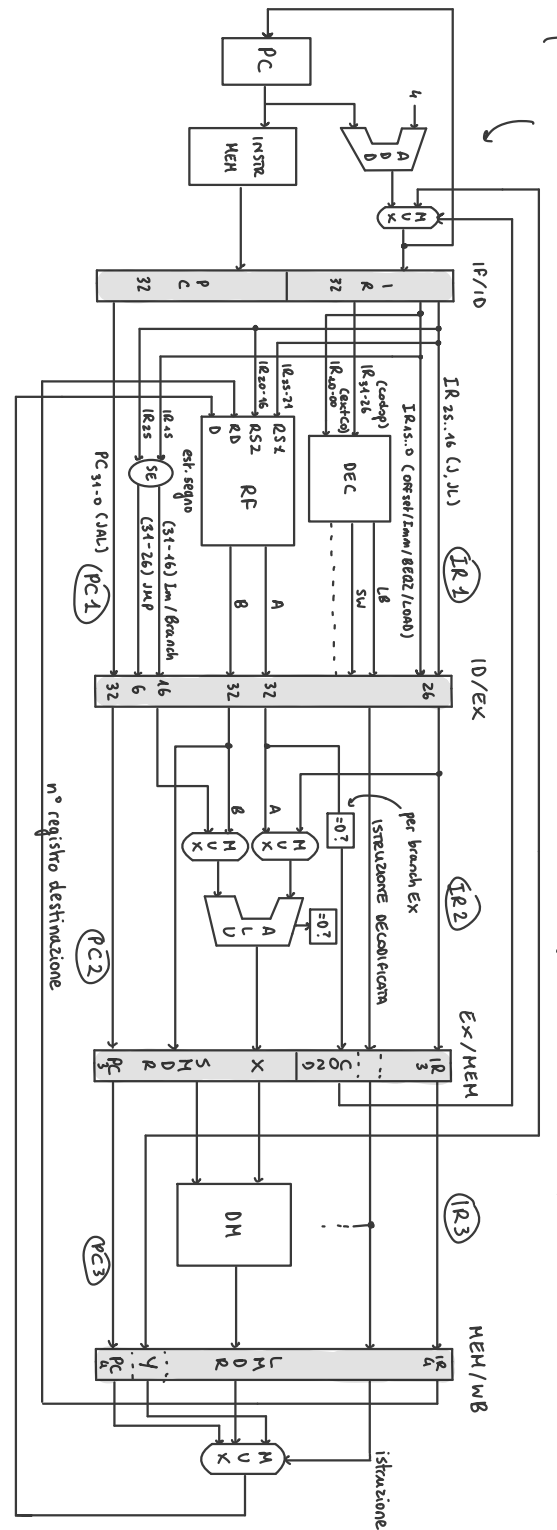
tra ciascuna attività sono presenti dei registri edge-triggered (pipeline registers) per mantenere le info necessarie

di conseguenza la frequenza massima → $T_{clk} = T_d + T_p + T_{su}$ (dove T_d è il ritardo a monte del registro e T_{su} il suo tempo di setup)

sia dati che info di controllo

unità di controllo distribuita nel datapath

questo MUX serve se istruzione in WB modifica RD e c'è un'istruzione in ID che legge dallo stesso registro → bisogna con dato aggiornato che sta a essere scritto nel RF



ID (FETCH)

ADD → PC + 4
MUX → gestisce l'eventuale presenza di salto o meno se è preso

PC fornisce indirizzo "sbagliato" perché il DUX sta già facendo il fetch dell'istruzione che non doveva entrare nella pipeline
→ scorre istruzione innanzi fino allo stadio di MEM (dove verrà eliminata con un NOP)

possiamo rimpiazzare lo schema basato su registro e mux con un counter a 30 bit (non a 32 perché gli ultimi 2 bit sono sempre a 0 (istruzioni a 32 bit e indirizzi allineati))
PC + 1 a 30 bit
PC + 4 a 32 bit

IF (DECODE)

- vengono estratti 2 registri ipotetici A e B con metodo dei campi fissi (S bit + S bit in posizione fissa 25-32)
- si porta PC + 4 nel caso di una JAL
- estensione del segno
↳ load e branch → I a 16 bit IR[15]
- ↳ jump → I a 26 bit IR[25]
- decodifica del codice op. IR[31-16] per capire di che istruzione si tratta
- decodifica IR[14:0] nel caso di istruzioni di tipo R (ultimi 11 bit dedicati all'extCod → estensione del codice operativo)

negli altri casi vengono oscurati nel decoder

nel DUX pipeline A e B non sono nel Register File ma all'esterno

il RF viene usato per estrarre i due indirizzi sorgente e dal WB per eventualmente sovrascrivere

EX (EXECUTE)

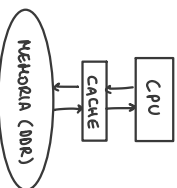
ALU → introduce un ritardo
↳ 0? per le istruzioni di SET (X"registro di uscita")
COND → si controlla se uno dei due registri è 0 (condizione per un salto)
↳ la condizione viene campionata nel registro cas quando passa nella fase di MEM viene retroscritta al MUX dello stadio IF
serve per pilotare la destinazione del salto data dal registro X
(ALU → registro + immediato)

MEM (MEMORY)

il branch vengono retroscritti al Fetch, molte istruzioni non fanno nulla
↳ l'indirizzo della memoria è stato calcolato nella EX, se il dato è nella cache l'accesso è rapidissimo
↳ sono necessari due MDR (LDR e SHDR) per gestire il caso di una load seguita immediatamente da una store
(WB-MEM sovrapposti → RF)
in ogni ciclo di clock devono poter essere eseguiti 2 accessi alla memoria (IF-MEM)

WB (WRITE-BACK)

viene terminata l'istruzione e si va ad aggiornare il registro destinazione
per vari motivi (letto qualcosa dalla memoria, JAL quindi va salvato il PC in R31, ALU, ecc...)
per scrivere nel registro destinazione servono i 5 bit dell'istruzione codificata a 32 bit presa dalla memoria
↳ l'informazione che voglio scrivere in questo registro sta nel RF

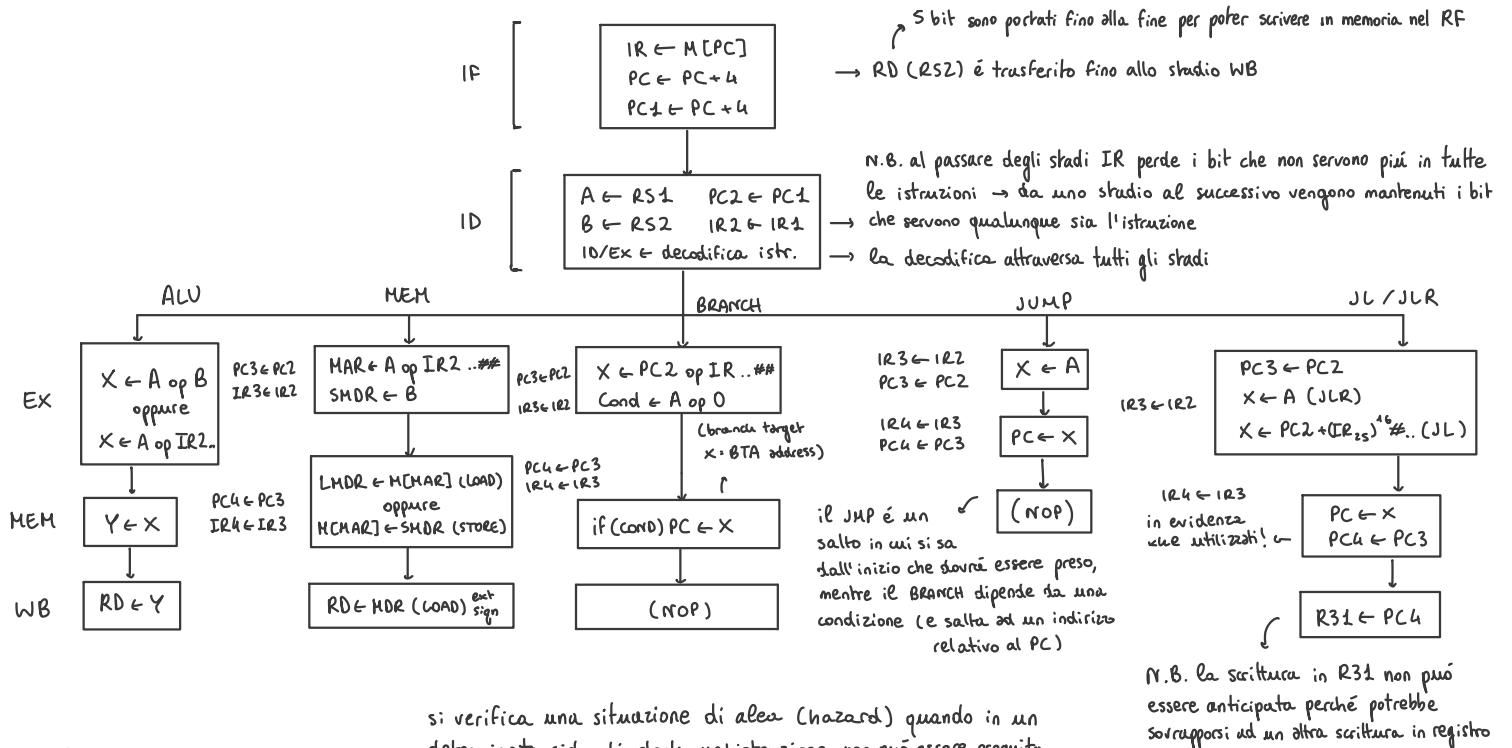


il clock della CPU è determinato dallo stadio più lento
+ veloci
+ piccoli

Architettura Harvard

Instruction Memory (IM)
Data Memory (DM)
memorie cache (on-chip)
(memorie prioritarie trasparenti a il processore)

ESECUZIONE IN PIPELINE DELLE ISTRUZIONI



ALOE / STALLI NELLE PIPELINE

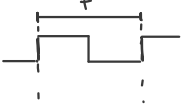
si verifica una situazione di aloe (hazard) quando in un determinato ciclo di clock un'istruzione non può essere eseguita

viene bloccata lei + quelle che la seguono → stallo nella pipeline → quelle che la precedono avanzano normalmente (rimozione causa aloe)

- Aloe strutturali** → una risorsa è condivisa fra due stadi della pipeline → istruzioni che si trovano contemporaneamente in tali stadi non possono essere eseguite simultaneamente → ad esempio stesse risorse in memoria → alcune le abbiamo risolte (2 ALU per IF e EX, 2 cache specializzate IM e DM)
- Aloe di dato** → dipendenze fra le istruzioni → RAW (Read After Write) → se ad esempio l'istruzione I_i necessita di un dato prodotto da I_{i-1} deve aspettare fino al WB della I_{i-1}
- Aloe di controllo** → le istruzioni che seguono un BRANCH dipendono dal loro risultato quindi c'è il rischio di far finire altrove nella pipeline istruzioni che non dovrebbero → problema capire se il branch è preso oppure no

GESTIONE DELLE ALOE DI DATO

- Forwarding** → consiste nel far avanzare le istruzioni nella pipeline anche se hanno prelevato dal register file il valore non aggiornato di un registro sorgente → una volta raggiunto lo stato di EX una rete logica detta **forwarding unit** si attiva per risolvere RAW senza far arrivare il dato corretto dalla posizione in cui si trova il dato

- Split cycle** → un clock viene diviso in 2 semi-periodi → fatto x evitare problemi quando WB modifica un indirizzo interno e ID ci preleva su

 - positivo → scrive il registro
 - negativo → legge il registro
 (non conveniente: obbliga a concludere le fasi di WB e ID nella metà del tempo)

- Aloe di dato** → dopo le LOAD possiamo avere uno stallo se il RD = IR dell'istruzione che segue → es. ADD R4, R1, R7
 LW R1, 32(R6)
 (il dato richiesto da ADD è presente solo nella fase di MEM → è necessario stallo)
 IF ID EX MEM WB
 IF ID S MEM WB
- Delayed Load** → gestita non via hardware con stallo ma via software → il compilatore riempie il delay slot dopo una LOAD con un'istruzione utile senza dipendenze di dato (nel caso peggiore con una NOP stallo)

GESTIONE ALOE DI CONTROLLO

- Stalli nel salto** → per evitare la proliferazione di dati sbagliati nella pipeline dopo un salto il DLX sostituisce i codici operativi delle 3 successive operazioni provenienti dalla memoria con il codice NOP → **svuotare la pipeline** → PC sa se va preso un salto solo in EX → rilevato in MEM
 (3 stalli)
- Always stall** → BRANCH → inserimento di 3 stalli a prescindere
 (viene bloccata la pipeline, la BRANCH avanza, riprende solo quando è in WB)
 (si può anticipare il rilevamento in EX → 2 stalli ma è una strategia aggressiva → problemi di timing)
- Predict not taken** → il DLX ipotizza il salto sempre come NOT TAKEN → le istruzioni dopo la branch vengono caricate e quando la BRANCH è in MEM se è TAKEN → NOP delle 3 istruzioni seguenti → nessuna di esse arriva al WB
- Delayed branch** → uguale alla delayed load + predict not taken → è il compilatore a inserire le NOP, non il processore!
- Branch target buffer** → approccio che cerca di predire già in IF se c'è un salto, se sarà taken e la sua destinazione
 (salva i jump eseguiti in precedenza → CACHE → indirizzo del salto, taken/not taken, indirizzo destinazione)
 (genera dei bit di predizione hit/miss che decidono il futuro della pipeline)
 (se 1 bit predizione e 2 loop annidati → funziona se viene eseguito + volte stesso salto (loop) → errore si usano 2 bit)