Appunti di Calcolatori Elettronici T

Esempi di architetture di elaborazione

- CPU (multicore / embedded)
- SOC (System on a Chip)
- GPU
- FPGA (Field Programmable Gate Array)

Tipologie di CPU e vantaggi/svantaggi

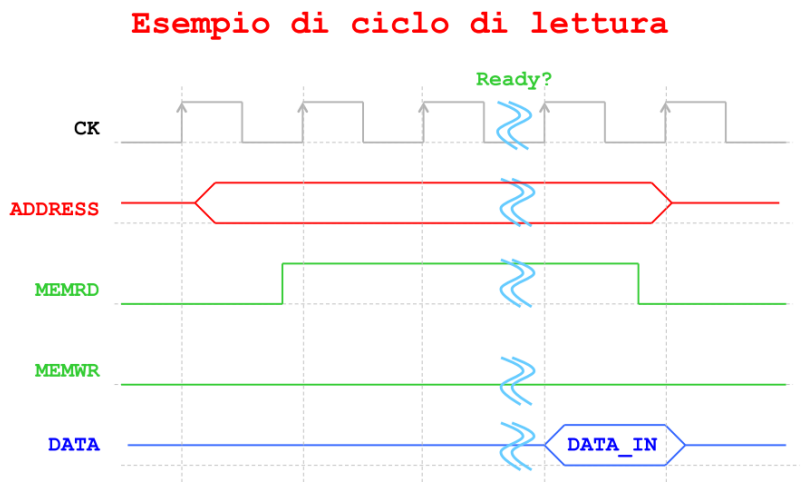
- RISC (Reduced Instruction Set Computer) -> ARM
- CISC (Complex Instruction Set Computer) -> Intel, AMD

Ogni singola istruzione RISC è eseguita spesso più rapidamente di una istruzione CISC dunque, il codice RISC, anche se più denso, è più veloce. Essendo le RL RISC più semplici lo spazio (silicio) può essere utilizzato per altre finalità (registri, cache, ecc.).
I processori RISC sono molto diffusi per dispositivi come smartphone mentre per i pc si usano ancora molto i RISC per via del software esistente anche se internamente i moderni processori CISC sono RISC.

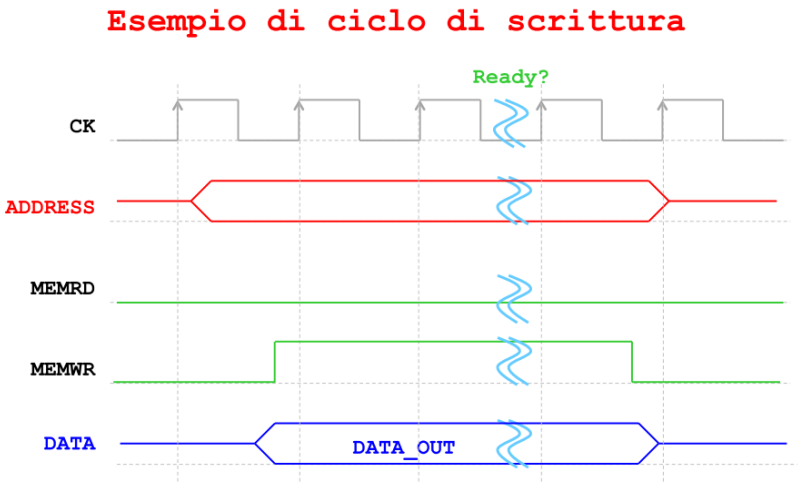
Segnali predefiniti

- Indirizzi BA[K..0]
- Dati BD[R..1]
- Controllo READ, WRITE

Ciclo di lettura



Ciclo di scrittura



Spazio di indirizzamento

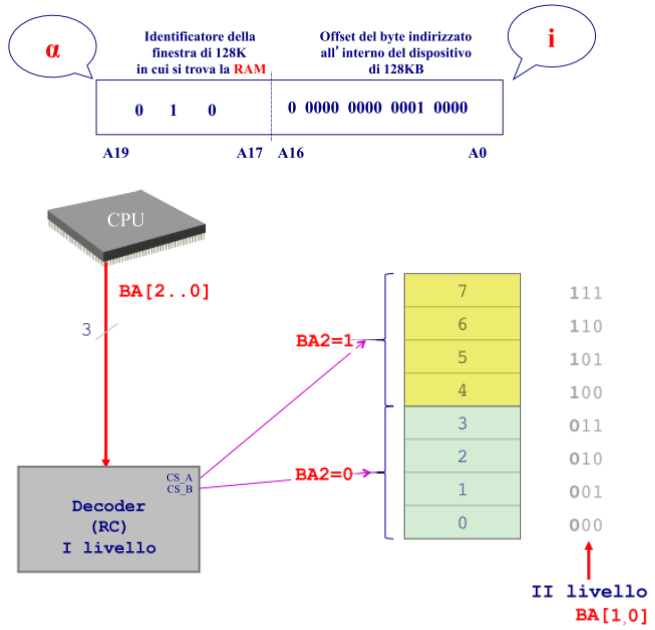
Il numero di diversi indirizzi emessi dalla CPU costituisce lo spazio di indirizzamento.
Una CPU che emette un indirizzo a 32 bit ha uno spazio di indirizzamento di 4 GB (2^32).

Condizione necessaria affinché un dispositivo fisico sia accessibile via software è che sia mappato in uno spazio di indirizzamento, cioè al dispositivo sia associata una finestra di indirizzi.
Un dispositivo accessibile attraverso il bus occupa in generale n = 2^k posizioni nello spazio di indirizzamento dove

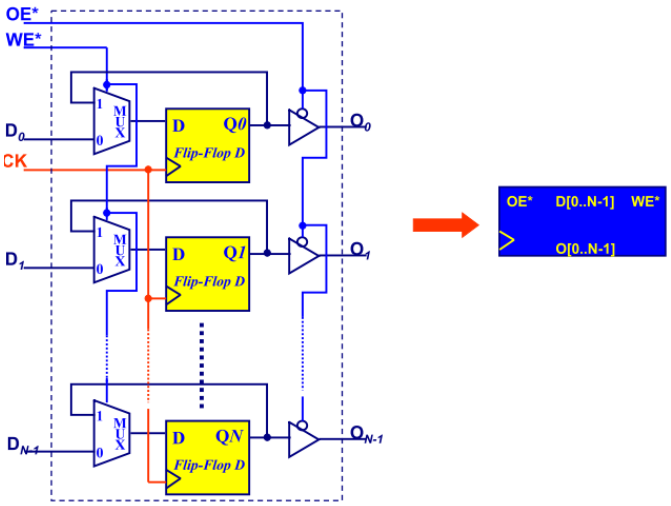
- n rappresenta il numero di byte indirizzabili all'interno del dispositivo
- k rappresenta il numero di bit di indirizzo interni al dispositivo

Per esempio per una RAM da n = 128 KB si ha k = $\log_2(n)$ = 17.
NB: K = 1024 byte

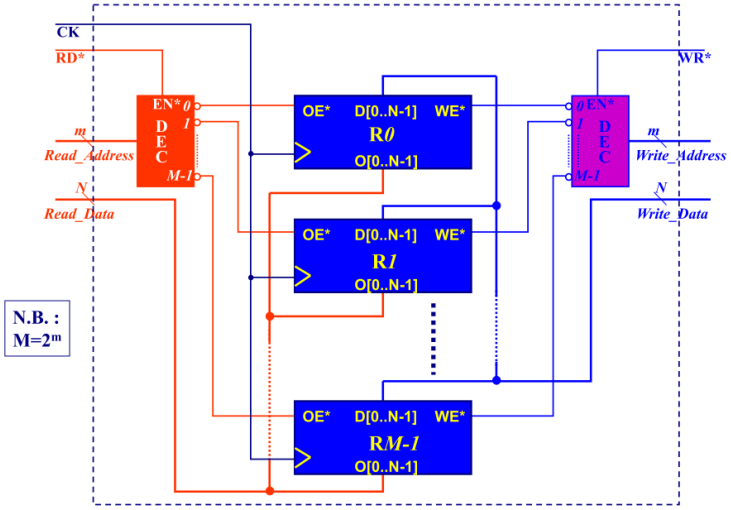
Si dice che D è mappato all'indirizzo A se gli indirizzi dei byte di D sono compresi tra A e A+(n-1).
Si dice che D è allineato se A è un multiplo di n (numero di bytes interni al dispositivo) cioè i k bit meno significativi di A sono uguali a zero.



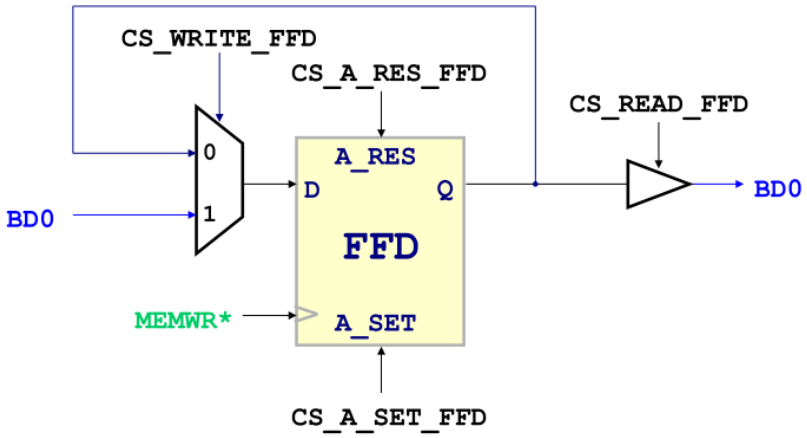
Registro Edge-Triggered con WE*



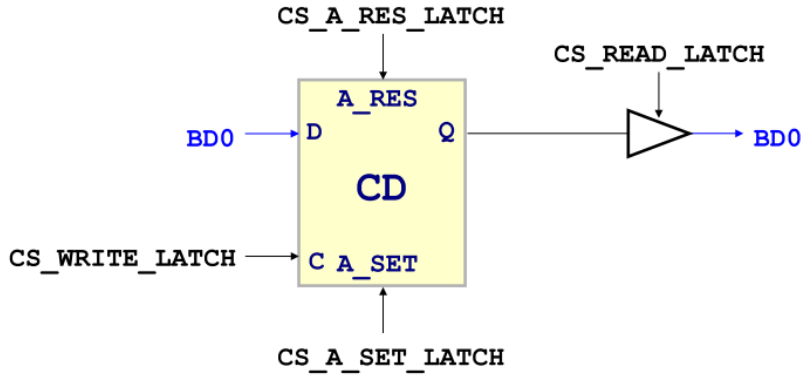
Register File (1 read-port, 1 write-port)



Mapping, read, write e set/reset di un FFD



Mapping, read, write e set/reset di un LATCH-CD



Linguaggio macchina e assembler

L'insieme delle istruzioni e dei registri di una CPU costituiscono l'Instruction Set Architecture (ISA).
Oltre alla possibilità di poter risolvere un qualsiasi problema, un requisito fondamentale di un linguaggio macchina/ISA è quello di minimizzare il tempo di esecuzione del codice:

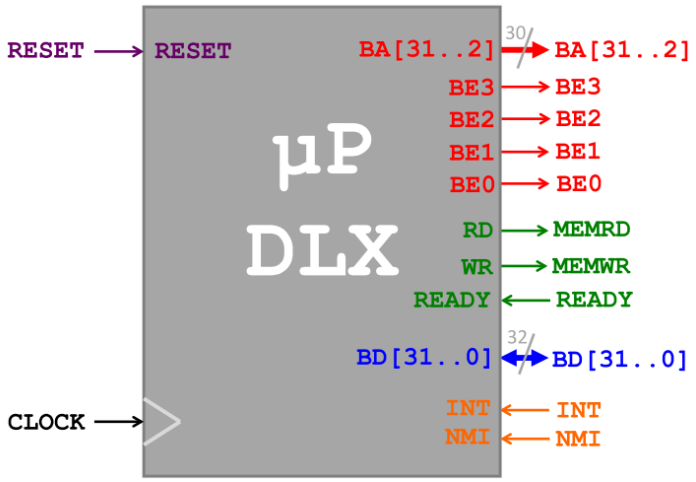
$$CPU_{Time} = N_{istruzioni} * CPI_{medio} * T_{CK}$$

Esistono CPU con codifica delle istruzioni a lunghezza:

- costante (e.g., 32 bit caso RISC DLX e molti altri)
- variabile da istruzione a istruzione (Intel X86)

La codifica delle istruzioni in linguaggio macchina è poco intuitiva per gli esseri umani. Nel linguaggio assembler si codificano le informazioni in un modo (un po') più intuitivo.

Segnali del processore DLX



Caratteristiche DLX

- Unico spazio di indirizzamento di 4GB
- 32 registri da 32 bit GP (R0, …, R31, con R0=0) accessibili direttamente da codice
- 4 registri (PC, IAR, MAR, MDR) non accessibili direttamente da codice
- Istruzioni di lunghezza costante (32) bit allineate
- Campi delle istruzioni di dimensioni/posizioni fisse
- 3 formati di istruzione: I, R, J
- Non ci sono istruzioni per gestire lo stack
- Per istruzioni che prevedono un indirizzo di ritorno (JAL/JALR) esso è salvato in R31
- Non esiste un registro di FLAG settato dalle istruzioni ALU. Le condizioni sono settate esplicitamente nei registri
- È presente un'unica modalità di indirizzamento in memoria (indiretto, mediante registro + offset)
- Le operazioni aritmetico/logiche sono eseguite solo tra registri (non tra registri e memoria)
- Esistono alcune istruzioni (MOVS2I e MOVI2S) per spostare dati tra registri GP e registri speciali e viceversa

Tipi di dato DLX

- BYTE (8 bit)
- HALF-WORD (16 bit)
- WORD (32 bit)

Formato delle istruzioni DLX

I

LOAD, STORE, BRANCH, A/L e Sx con immediato, JUMP (con o senza ritorno) con registri

6 bit	5 bit	5 bit	16 bit
OpCode	**RS2 / Rd**	**RS1**	**Imm_{16}**

R

A/L e Sx con registri

6 bit	5 bit	5 bit	5 bit	11 bit
OpCode	**RS2**	**RS1**	**Rd**	**OpCode ext.**

J

JUMP (con o senza ritorno) con immediato

6 bit	26 bit
OpCode	**Immediato / offset relativo al PC**

Modalità di accesso alla memoria

I due principali metodi di accesso alla memoria (indirizzamento) sono:

- Diretto (indirizzo cablato nell'istruzione) es: LB R7, F800h
- Indiretto (indirizzo modificabile a run-time) es: LB R7, F800h(R3)

Il DLX prevede solo la modalità di indirizzamento indiretto.

Salti condizionati

Con una istruzione di tipo set seguita da un'istruzione di branch si realizza la funzione di compare and branch senza bisogno di flag.

Interruzioni

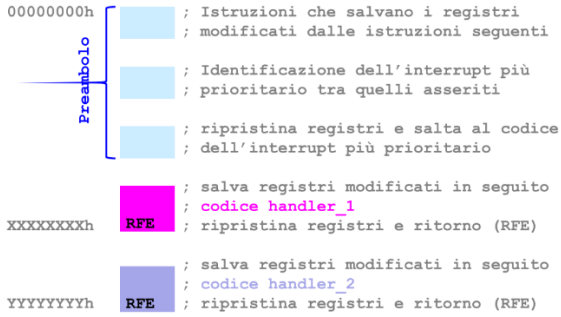
- Polling: strategia poco efficiente, consiste nel controllare periodicamente se si sono verificati eventi
- Interrupt: strategia molto più efficiente, basata su strategia push, cioè è il dispositivo a segnalare l'evento.

Un interrupt è un evento che interrompe la CPU durante il regolare flusso di esecuzione del codice.
Se la CPU è abilitata a ricevere tale segnalazione, esegue automaticamente una porzione di codice denominata interrupt handler al fine di gestire l'evento. Il codice deve terminare con l'istruzione RFE che riporta il controllo al chiamante.

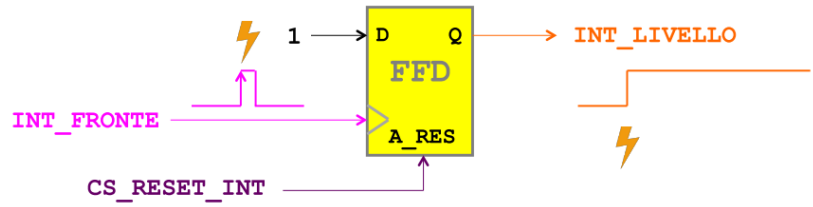
Gli eventi possono essere:

- Hardware — relativi a fattori esterni (es: premuto un tasto)
- Software — generati da programmi per interfacciarsi con il sistema (system calls)
- Exception — relativi a fattori interni (es: divisione per zero, overflow, …)

L'interrupt può verificarsi in qualsiasi momento e non è sincronizzato con il clock.
Proprio per questo motivo è importante salvare i registri prima dell'esecuzione dell'handler e ripristinare poi i registri modificati, in modo da mantenere la consistenza dei dati:



Nel caso del DLX la CPU è sensibile al livello del segnale.
Nel caso dei dispositivi che generano interrupt, questo rimane a 1 fintantoché la causa che lo ha generato non sia stata gestita.
Se così non fosse è necessario portare il segnale di interrupt a livello con un FFD e resettarlo a 0 via software mediante un opportuno comando software che asserisce il reset:



Gestione di interruzioni multiple e priorità

Il DLX ha un solo segnale di interrupt. Per gestire sorgenti multiple tipicamente si convogliano mediante gate tutti gli interrupt verso l'unico segnale di interrupt.

1. PROBLEMATICA: Come determinare quali interrupt sono asseriti in un determinato istante e qual è l'ordine in cui vanno gestiti?

   È necessario poter determinare lo stato delle richieste di interrupt mediante opportune istruzioni software, in particolare esistono reti denominate PIC che agevolano il compito alla CPU.
2. PROBLEMATICA: Come fare nel caso in cui mentre si sta eseguendo un handler arriva un'interruzione più prioritaria?

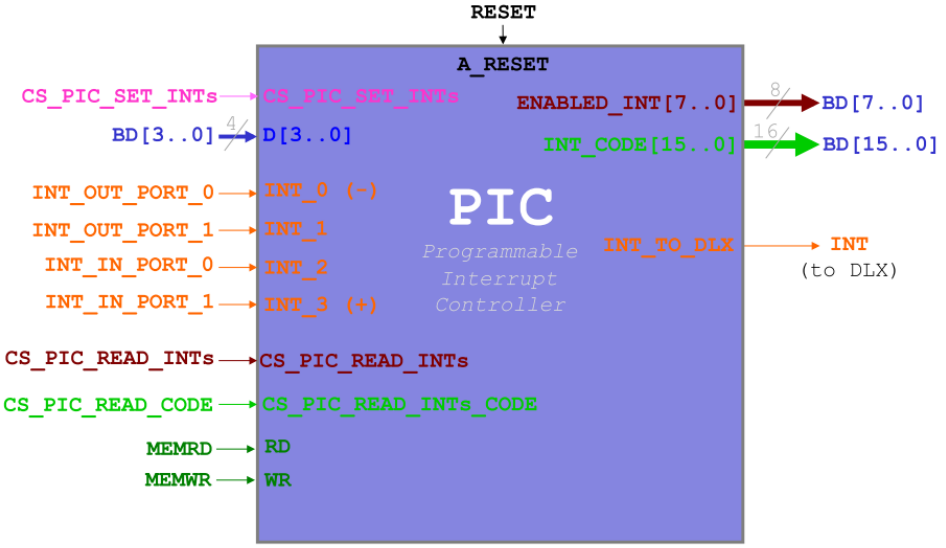
   Si parla in questo caso di **nesting** o annidamento delle interruzioni. Il DLX base non permette di interrompere l'interrupt handler per dare il controllo all'interrupt più prioritario, sarebbe opportuno gestirlo con uno stack software e avere la possibilità di riabilitare gli interrupt nell'handler mediante opportune istruzioni non previste nell'ISA base.

PIC

Il PIC (Programmable Interrupt Controller) si occupa di gestire multiple sorgenti di interruzione e di fornire direttamente alla CPU (su richiesta) qual è il codice/indirizzo dell'interrupt più prioritario tra quelli asseriti in quel momento evitando che sia quest'ultima a dover interrogare le singole periferiche.
Tipicamente, in un PIC è possibile disabilitare le singole sorgenti di interruzione e stabilire il livello di priorità di ciascuna in accordo a varie politiche (priorità fissa, variabile, ecc.)

Esempio di PIC realizzato a lezione che permette di:

- Indicare le porte abilitate (CS_PIC_SET_INTs e D[3..0])
- Conoscere gli interrupt, tra quelli abilitati, asseriti in quel momento (CS_PIC_READ_INTs e ENABLED_INT[7..0] di cui 4 bit sono cablati a 0)
- Conoscere il codice dell'interrupt più prioritario asserito in quel momento (CS_PIC_READ_CODE e INT_CODE[15..0])



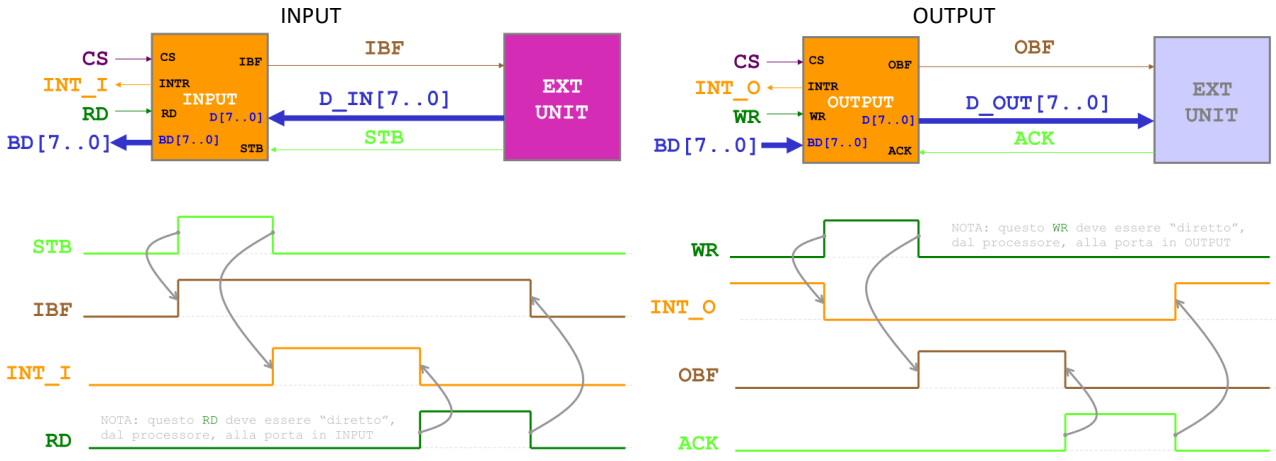
NMI

Il segnale NMI (Not Maskable Interrupt) è un ulteriore segnale in input al quale sono collegate un numero limitato di sorgenti di interruzioni particolarmente critiche (es: segnale di imminente perdita di alimentazione elettrica).
Una richiesta di interrupt inviata sul pin NMI non può essere ignorata e interrompe l'esecuzione di altri handler.

Per la gestione del segnale NMI è necessario inserire le istruzioni nella prima parte del "preambolo" all'indirizzo 00000000h, prima di gestire gli interrupt che sono inviati attraverso INT.

Handshake

L' handshake è un approccio efficiente e ampiamente utilizzato per risolvere problemi di sincronizzazione tra produttore e consumatore (nel nostro caso CPU e unità esterna).

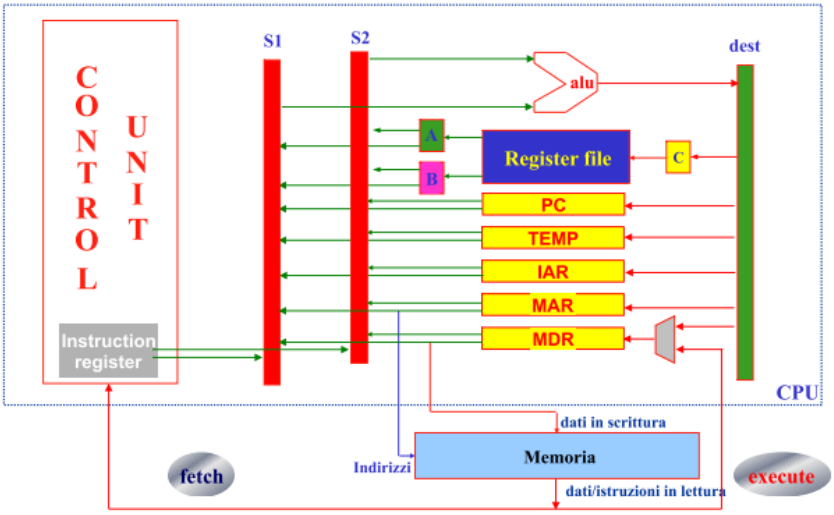


Datapath
Contiene tutte le unità di elaborazione ed i registri necessari per l'esecuzione delle istruzioni della CPU.

Micro-operazione
Operazione eseguita all'interno del datapath in un ciclo di clock.

Unità di Controllo
È una RSS che in ogni ciclo di clock invia un ben preciso insieme di segnali di controllo al datapath al fine di specificare l'esecuzione di una determinata micro-operazione

Struttura interna del DLX sequenziale



- Register File: 32 registri GP con R0 = 0
- PC (Program Counter) Indirizzo di memoria dell'istruzione corrente
- TEMP: Registro di deposito temporaneo di risultati
- IAR (Interrupt Address Register): Deposito dell'indirizzo di ritorno in caso di interruzione
- MAR (Memory Address Register): Indirizzo del dato da scrivere / leggere in memoria
- MDR (Memory Data Register): Registro di transito temporaneo di dati da e per la memoria
- IR (Instruction Register): Contiene l'istruzione attualmente in esecuzione

Funzioni della ALU

S1 + S2
S1 – S2
S1 and S2
S1 or S2
S1 exor S2
Shift S1 a sinistra di S2 posizioni
Shift S1 a destra di S2 posizioni
Shift S1 aritmetico a destra di S2 posizioni
S1
S2
0
1

Flag di uscita

Zero
Segno negativo
Carry

Clock minimo e frequenza massima

$$T_{CK} > T_C + T_{OE} + T_{ALU} + T_{SU}$$

- T_C: ritardo stabilizzazione segnali di controllo
- T_{OE}: tempo caricamento dati in A e B
- T_{ALU}: tempo esecuzione istruzione più lenta
- T_{SU}: tempo di set-up minimo per i registri

$$f_{CK}(MAX) = \frac{1}{T_{CK}}$$

Passi dell'esecuzione delle istruzioni

1. FETCH
   Istruzione prelevata da memoria e posta in IR
   $MAR \leftarrow PC$
   $IR \leftarrow M[MAR]$
2. DECODE
   Istruzione in IR viene decodificata e nel frattempo si prelevano gli operandi sorgente e si incrementa il PC
   $A \leftarrow RS1$
   $B \leftarrow RS2$
   $PC \leftarrow PC + 4$
3. EXECUTE
   Elaborazione aritmetica o logica mediante la ALU
   - LOAD
     $MAR \leftarrow A + (IR_{15})^{16} \# \# IR_{15\ldots0}$
   - STORE
     $MAR \leftarrow A + (IR_{15})^{16} \# \# IR_{15\ldots0}$
     $MDR \leftarrow B$
   - ALU
     $C \leftarrow A\ op\ B$ / $C \leftarrow A\ op\ (IR_{15})^{16} \# \# IR_{15\ldots0}$
     $C \leftarrow sign(A\ op\ B)$ / $C \leftarrow sign(A\ op\ (IR_{15})^{16} \# \# IR_{15\ldots0})$
   - JMP / JAL
     $TEMP \leftarrow PC + (IR_{25})^{6} \# \# IR_{25\ldots0}$
   - JR / JALR
     $TEMP \leftarrow A$
   - BRANCH
     $TEMP \leftarrow PC + (IR_{15})^{16} \# \# IR_{15\ldots0}$
4. MEMORY
   Accesso alla memoria e, nel caso di BRANCH aggiornamento del PC
   - LOAD
     MDR ← M[MAR]
   - STORE
     M[MAR] ← MDR
   - JAL / JALR
     C ← PC
   - BRANCH
     IF(Cond): PC ← TEMP
5. WRITE-BACK
   Scrittura sul Register File
   - JUMP
     PC ← TEMP
     RD ← C
   - LOAD / STORE / ALU / BRANCH
     C ← MDR
     RD ← C

DLX pipelined

Requisiti:

- Ogni stadio deve essere attivo in ogni ciclo di clock
- È necessario incrementare PC in IF anziché in ID
- I Pipeline Registers trasportano sia dati che informazioni di controllo (l'unità di controllo è distribuita fra gli stadi della pipeline).

Architettura Harvard

In ogni ciclo di clock devono poter essere eseguiti 2 accessi alla memoria. Per questo vengono predisposte due memorie cache

- IM (Instruction Memory)
- DM (Data Memory)

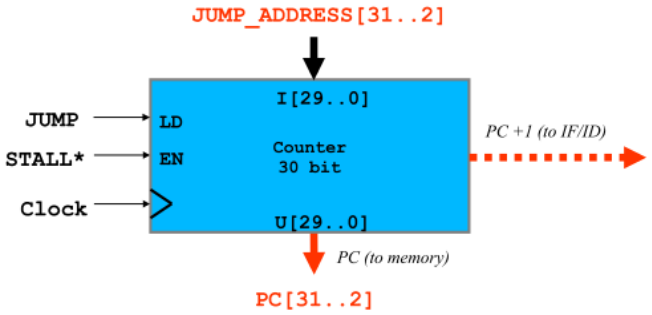
spesso strutturate in più livelli di grandezza per velocizzare l'accesso ai dati e evitare l'accesso alla memoria vera e propria.

Stallo

Blocco del clock dello stadio e di tutti quelli precedenti e propagazione progressiva agli stadi successivi.

Alee nelle pipeline

- Alee strutturali: Una risorsa è condivisa fra due stadi della pipeline: le istruzioni che si trovano correntemente in tali stadi non possono essere eseguite simultaneamente.
  Soluzione:
  a) Nel caso di ALU contesa tra IF e EX si introduce un contatore a 30 bit, che fa rifferimento ai 30 bit più significativi del PC, di modo da eseguire PC+4 semplicemente incrementando tale contatore e evitando di utilizzare l'ALU.



  b) Nel caso di memoria contesa tra IF e MEM si usa la strategia data del modello dell'**architettura Harvard**
- Alee di Dato: Sono dovute a dipendenze fra le istruzioni. Ad esempio una istruzione che legge un registro scritto da un'istruzione precedente (Read After Write).
  Soluzione:
  a) In generale: **Forwarding** o **Split Cycle**
  b) Per le alee di dato di LOAD si preferisce il **Delayed Load** perché tali alee possono indurre ritardi causati dall'accesso in memoria.
- Alee di Controllo: Le istruzioni che seguono un branch dipendono dal risultato del branch (taken/not taken).
  Soluzione:
  a) Stalli:
    I. Si retroaziona la destinazione del salto dalla fase di EX su un mux, condizionato da se il salto è stato preso o no, insieme al PC della successiva istruzione. L'uscita del mux è collegata alla cache di IM. Si forza una sola operazione vuota dopo il fetch dell'istruzione di salto.
    II. Si retroaziona la destinazione del salto dalla fase di EX e si forzano operazioni vuote negli stati IF e ID.
    III. Si forzano operazioni vuote (NOP) nei successivi tre stati (**Always Stall**).
  b) **Predict not taken**: Si continuano ad eseguire le 3 istruzioni successive che in caso in cui il salto risulta taken vengono cambiate dall'unità di controllo in NOP e si esegue il codice all'indirizzo del salto.
  c) **Delayed branch**.
  d) Dynamic Prediction con **Branch Target Buffer**.

Forwarding
Il forwarding si implementa inserendo dentro la pipeline una semplice rete logica che si occupa di controllare se il valore di alcuni registri è stato modificato dalla/e istruzioni precedenti e in tal caso viene retroazionato il dato aggiornato.
Nel caso di istruzione di LOAD però si preferisce stallare la pipeline per un clock o inserire una istruzione utile (Delayed Load) per via di eventuali e inevitabili ritardi sulla lettura del dato.

Split Cycle
Permette di eliminare le alee di dato (RAW) in alternativa al MUX davanti al RF. Consiste nello scrivere sul RF nel primo semiperiodo e leggere in esso nel secondo semiperiodo. In questo modo l'istruzione che si trova in ID non avrà mai registri obsoleti, perché sono stati aggiornati dalla fase di WB nel semiperiodo precedente. Lo svantaggio è che la frequenza del RF va raddoppiata e ciò può causare malfunzionamenti del RF.

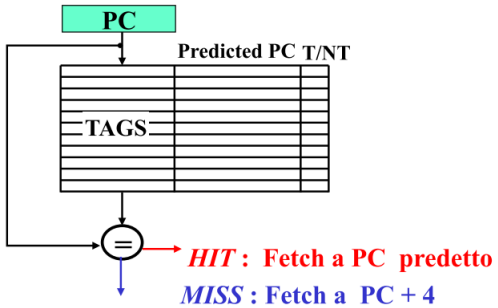
Delayed Load
In diverse CPU RISC l'alea associata alla LOAD non è gestita in HW stallando la pipeline ma è gestita via SW dal compilatore che cerca di riempire il delay-slot con un'istruzione utile o nel caso peggiore con NOP.

Istruzione LOAD
delay slot
Istruzione Successiva

Delayed Branch
In diverse CPU RISC l'alea associata alle istruzioni di BRANCH non è gestita in HW stallando la pipeline ma è gestita via SW dal compilatore che cerca di riempire i delay-slot con istruzioni utili o nel caso peggiore con NOP.

Istruzione BRANCH
delay slot
delay slot
delay slot
Istruzione Successiva

Branch Target Buffer
Tecnica di predizione dinamica che sfrutta una piccolissima cache per memorizzare le istruzioni di BRANCH man mano che si incontrano, l'indirizzo a cui si chiede di saltare e se l'ultima volta essa risultava taken o not taken. Per ogni BRANCH si verifica se essa è già presente nella cache, se così fosse si decide se saltare o meno in base alla tabella.
Nel caso di predizione corretta: 0 stalli
Nel caso di predizione sbagliata: da 1 a 3 stalli



Questa tecnica permette di ottimizzare molto l'esecuzione della pipeline in caso di cicli.
Si preferisce in genere usare 2 bit rispetto a 1 per la decodifica di T/NT per ottimizzare ancora di più l'esecuzione della pipeline in caso di più cicli innestati.