ARCHITETTURA RISC

CISC: complex instruction set computer

Insieme di istruzioni ricco, con un gran numero di istruzioni complesse, motivato dai seguenti fattori:

- diminuire il numero di istruzioni ISA per ciascuna istruzione di alto livello in modo da poter semplificare i compilatori,
- ottenere programmi più brevi e veloci,
- realizzazione efficiente delle istruzioni complesse in microcodice.

RISC: reduced instruction set computer

Insieme di istruzioni ridotto, formato da istruzioni relativamente semplici, realizzato avendo in mente i principi guida descritti nel seguito e le conseguenze che comportano.
Completare, se possibile, un'istruzione per ciclo macchina (prelievo di due operandi da registri, operazione sull'ALU, memorizzazione del risultato in un registro). Utilizzo di istruzioni semplici che possono essere eseguite direttamente in HW con velocità superiore, dato che non richiedono microcodice e il relativo HW.

Esecuzione da registro a registro, la maggior parte delle operazioni coinvolge operandi presenti nei registri, l'accesso alla memoria avviene solo attraverso istruzioni load e store. Per un dato tipo di istruzione (p.e. somma aritmetica) sono necessarie poche varianti, l'HW risulta più semplice, l'esecuzione più veloce (i registri sono il tipo di memoria più veloce in assoluto): dato che nella pratica gli accessi alla memoria risultano fortemente localizzati, l'utilizzo di un elevato numero di registri e/o l'ottimizzazione nel loro uso limita grandemente i lenti trasferimenti da e verso la memoria principale.

Semplici modi di indirizzamento, quasi tutte le istruzioni utilizzano l'indirizzamento tramite registri. Istruzioni e HW risultano più semplici.

Semplici formati delle istruzioni, lunghezza delle istruzioni fissata e allineata sui confini di parola, posizione dei campi all'interno dell'istruzione (p.e. opcode) fissa. Istruzioni semplici e regolari permettono di avere l'HW di decodifica semplice e veloce, e ottimizzare il fetch delle istruzioni.

Le semplificazioni introdotte portano vantaggi in termini di progettazione e realizzazione dell'HW.

Dal punto di vista delle prestazioni alcuni possibili vantaggi sono

- maggior facilità nello sviluppo di compilatori ottimizzanti: risulta più facile utilizzare e organizzare in maniera efficiente istruzioni semplici e primitive, più semplice massimizzare l'uso dei registri
- l'HW è ottimizzato per eseguire in maniera efficiente istruzioni semplici, e le istruzioni più frequentemente utilizzate nei programmi risultano essere quelle relativamente semplici
- maggiore efficienza nell'utilizzo delle pipeline

Le caratteristiche tipiche di un RISC classico sono:

- formato fisso delle istruzioni,
- dimensione tipica di 32 bit,

- pochi modi di indirizzamento,
- assenza dell'indirizzamento indiretto,
- assenza di istruzioni che combinano accessi alla memoria e operazioni aritmetiche,
- assenza di istruzioni con più di un operando in memoria,
- nessun supporto all'allineamento arbitrario di dati per le operazioni di load e store,
- massimo utilizzo dell'unità di gestione della memoria,
- almeno 5 bit per identificare i registri interi,
- almeno 4 bit per identificare i registri in virgola mobile.

PIPELINE RISC

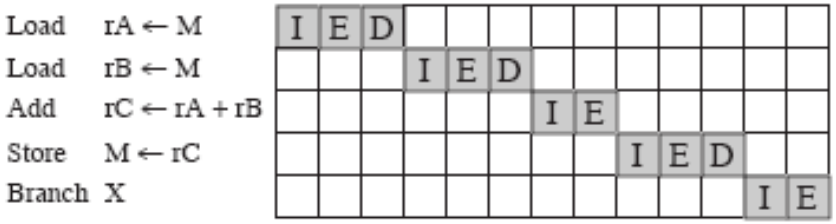
Molto schematicamente possiamo pensare di suddividere la pipeline RISC nei seguenti stadi,

- per le istruzioni che non fanno riferimento alla memoria
  - I: fetch dell'istruzione
  - E: esecuzione (ALU e I/O dai registri)
- per le istruzioni che fanno riferimento alla memoria (load/store)
  - I: fetch dell'istruzione
  - E: esecuzione (calcolo dell'indirizzo)
  - D: memoria (I/O registro-memoria)

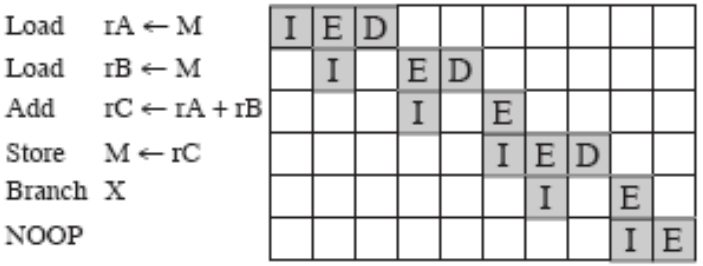
Più realisticamente lo stadio E potrebbe essere a sua volta suddiviso in più stadi, p.e. E_1 ed E_2 in modo da bilanciare meglio la durata dei singoli stadi.
Nella figura seguente sono schematizzate diverse realizzazioni possibili per una pipeline, in ordine di efficienza crescente, con riferimento all'esecuzione di una sequenza di istruzioni

- (a): esecuzione sequenziale
- (b): esecuzione con pipeline in grado di gestire simultaneamente gli stadi I ed E di due diverse istruzioni (con la possibilità di un solo accesso in memoria per stadio)

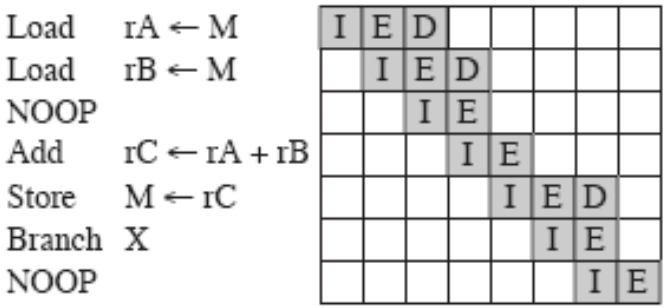
- (c): esecuzione con pipeline in grado di gestire simultaneamente gli stadi I, E e D di diverse istruzioni (con la possibilità di due accessi indipendenti in memoria per stadio, p.e. accesso indipendente a istruzioni e dati)
- (d): come (c) ma con lo stadio E diviso in due stadi indipendenti E_1 ed E_2



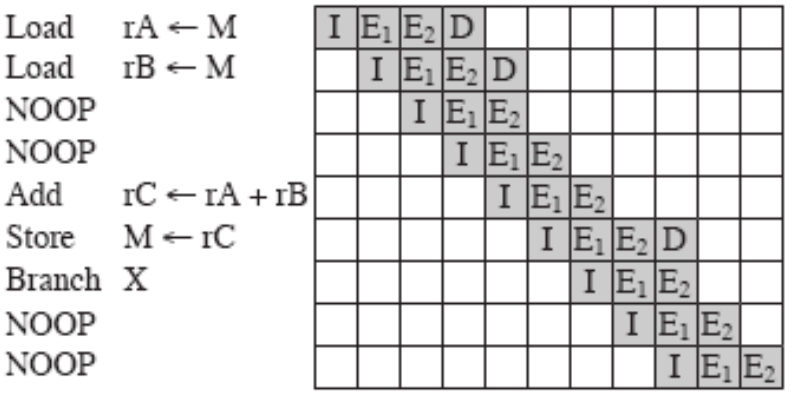
(a) Sequential execution



(b) Two-stage pipelined timing



(c) Three-stage pipelined timing



(d) Four-stage pipelined timing

Un esempio reale: la pipeline del processore R3000 è costituita dai seguenti 5 stadi

- fetch dell'istruzione,
- lettura dei registri sorgente,
- operazione ALU o calcolo dell'indirizzo di memoria,
- accesso alla memoria,
- scrittura del registro destinazione.

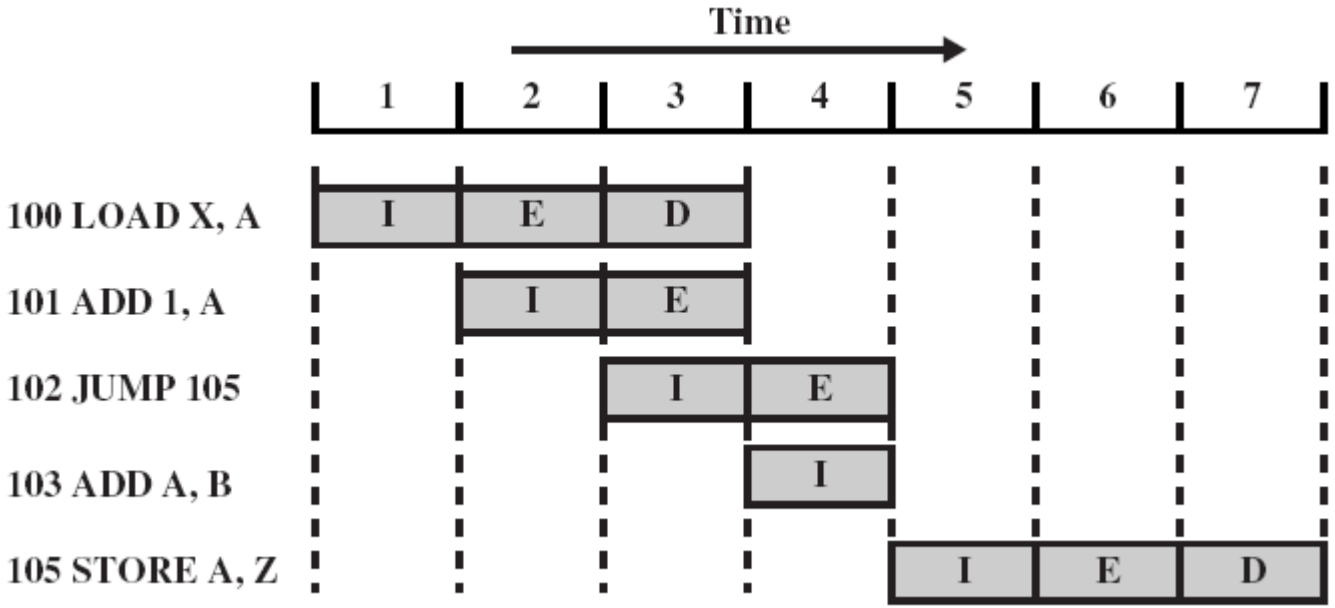
DELAY SLOT

La natura semplice e regolare delle istruzioni RISC permette di utilizzare e ottimizzare il funzionamento della

pipeline. In particolare consideriamo il caso dei salti non condizionati per introdurre il concetto di delay slot.
Nella tabella seguente è rappresentata una sequenza di istruzioni contenti un salto (JUMP 105).

indirizzo	istruzione	
100	LOAD	X,A
101	ADD	1,A
102	JUMP	105
103	ADD	A,B
104	SUB	C,B
105	STORE	A,Z

L’utilizzo “normale” della pipeline durante l’esecuzione di tali istruzioni è rappresentato nella figura seguente. Notare come, dopo l’istruzione di salto, sia necessario svuotare la pipeline dalle istruzioni “indesiderate”.

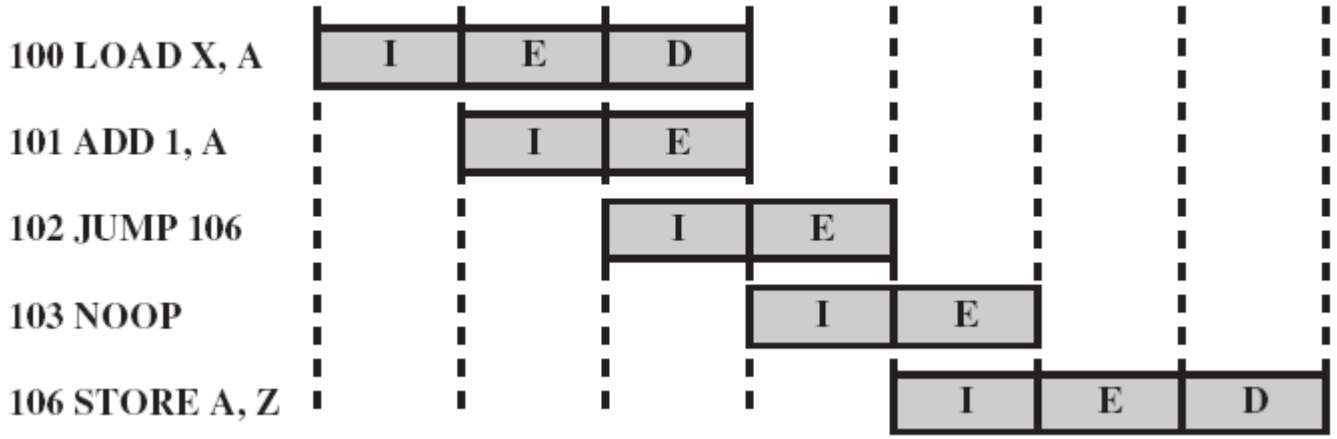


L’approccio RISC prevede l’utilizzo del salto ritardato (delayed branch), ovvero di un salto che non ha effetto fino al termine dell’istruzione seguente, in modo tale che l’istruzione immediatamente successiva a quella di salto viene sempre eseguita. La locazione dell’istruzione immediatamente successiva al salto prende il nome di delay slot. Inserendo un’istruzione di no-operation (NOOP) nel delay slot si regolarizza la pipeline, e non è necessario svuotarla.

indirizzo	istruzione
100	LOAD X,A

101	ADD	1,A
102	JUMP	106
103	NOOP	
104	ADD	A,B
105	SUB	C,B
106	STORE	A,Z

L’utilizzo della pipeline durante l’esecuzione del salto ritardato con l’istruzione NOOP nel delay slot è rappresentato nella figura seguente.

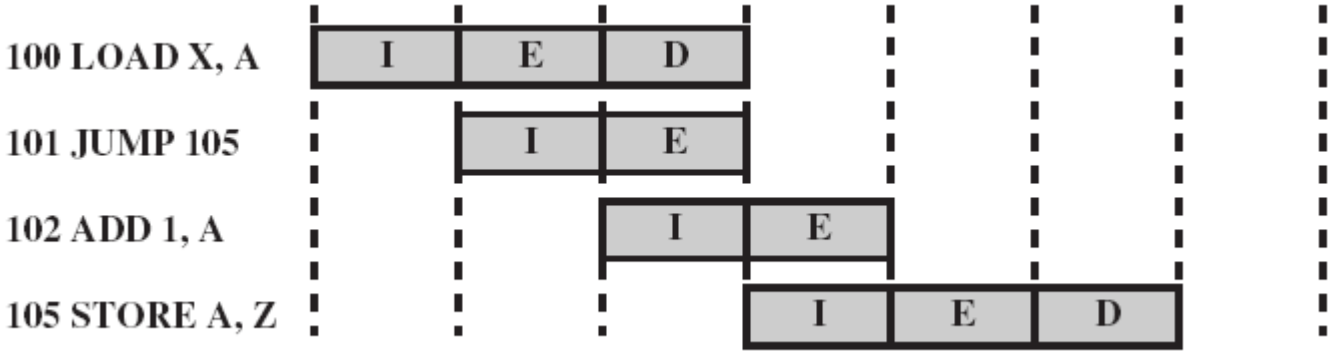


Per ottenere un miglioramento nelle prestazioni è possibile invertire le istruzioni 101 e 102 (che non dipendono l'una dall'altra), sostituendo la NOOP nel delay slot con la ADD, come illustrato in tabella.

indirizzo	istruzione

100	LOAD	X,A
101	JUMP	105
102	ADD	1,A
103	ADD	A,B
104	SUB	C,B
105	STORE	A,Z

L’utilizzo della pipeline è rappresentato nella figura seguente, dove osserviamo che abbiamo risparmiato un ciclo di clock.



La tecnica del riempimento del delay slot con l'istruzione immediatamente precedente al salto può essere utilizzata nel caso di salti non condizionati e chiamate a procedura, mentre nel caso di salti condizionati è necessario prestare attenzione alle dipendenze fra istruzioni (p.e. quando la condizione dipende dall'istruzione precedente al salto che si vorrebbe portare nel delay slot). Anche nel caso

dell’istruzione di load si può applicare una tecnica simile (delayed load), in modo da fare del lavoro utile in attesa che il trasferimento dalla memoria venga completato.