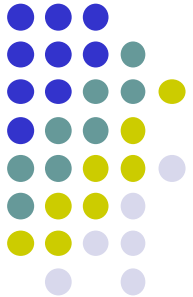
Reti a più stadi



- Molto spesso nella progettazione digitale può essere necessario suddividere la rete in un certo numero di stadi operanti in cascata
- Tali stadi possono essere disaccoppiati e sincronizzati in modo da operare in istanti di clock successivi interponendo flip-flop
- In questo caso, i flip-flop agirebbero da *registri di transito*, sfruttando il modello di Moore

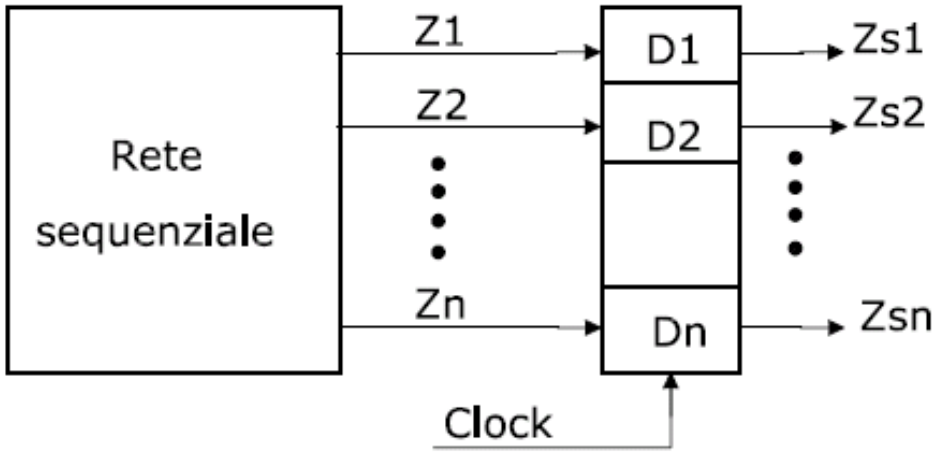


Figura 4.27 Sincronizzazione delle uscite tramite interposizione di un registro di transito.

- Lo schema della suddivisioni in stadi in cascata è dunque il seguente

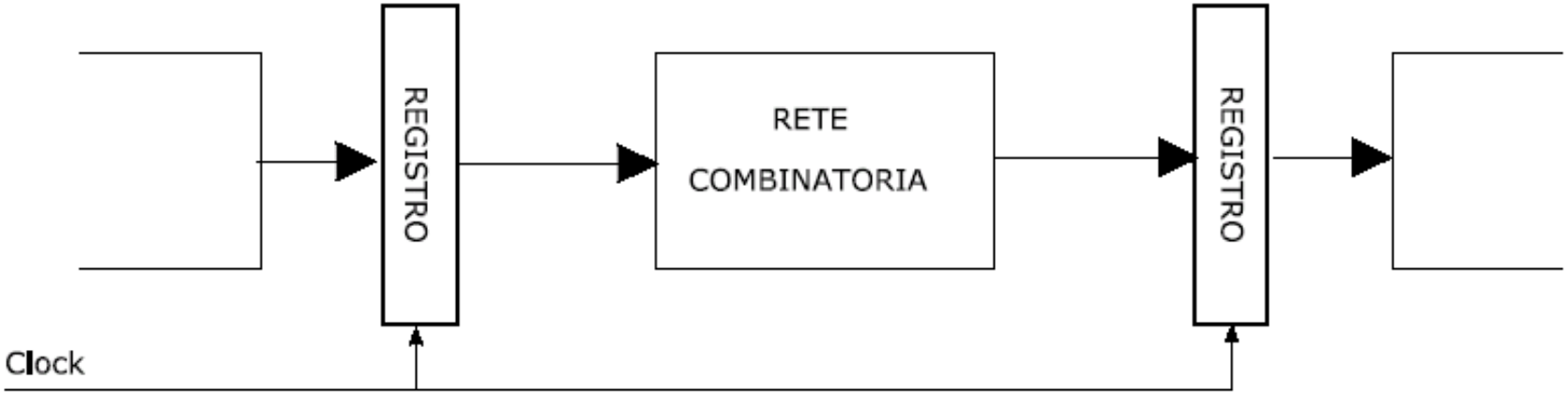
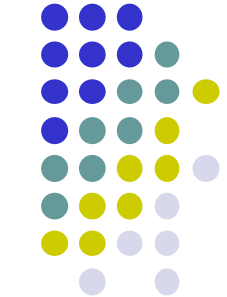


Figura 4.28 Modello di rete di elaborazione. La rete combinatoria deve elaborare i segnali in ingresso e presentarli al registro in uscita in modo che vengano memorizzati sul prossimo fronte (di commutazione) del clock.

Registri



- Un registro può essere definito come un insieme di *n* elementi di memoria (flip-flop) identici e sincronizzati da un unico clock
- Sono gli elementi dei calcolatori che mantengono le informazioni
- Permettono di partizionare la logica complessiva e scomporla in blocchi semi-indipendenti
- Possono differenziarsi per i tipi di flip-flop utilizzati, per il modo in cui vengono comandati e per le funzioni aggiuntive (scorrimento, conteggio, ecc.)
- Vediamo l'esempio seguente

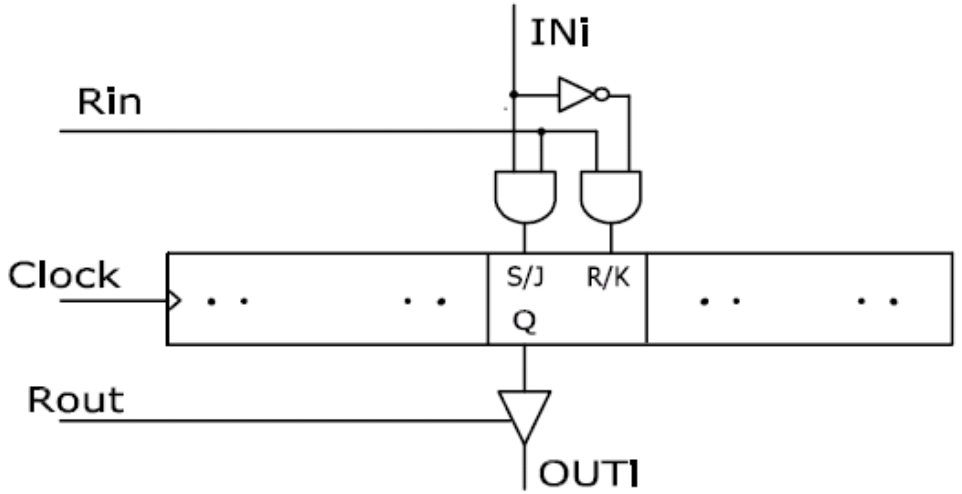


Figura 4.29 Struttura di un registro. Viene mostrato il dettaglio per il generico bit. Si assume che il registro commuti sul fronte di discesa del clock. Il segnale R_{out} abilita l'uscita in terzo stato.

- Se R_{in}=*0*, l'ingresso ai flip flop è *00* e quindi il registro mantiene lo stato precedente
- Se R_{in}=*1*, lo stato (il contenuto) del flip-flop *i* diventa quello corrispondente all'ingresso IN_i
- R_{out} ha funzione di Output Enable dell'uscita a tre stati
- R_{in} e R_{out} svolgono funzioni di controllo o di comando

- D'ora in poi, salvo avviso contrario, assumeremo che i registri commutino sul fronte di discesa del clock

- Lo schema semplificato generale usato nel seguito è il seguente

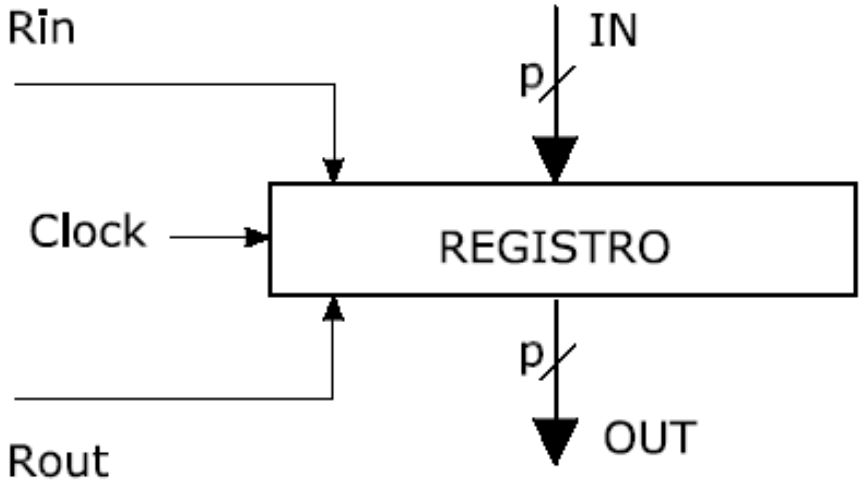
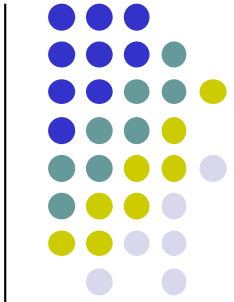


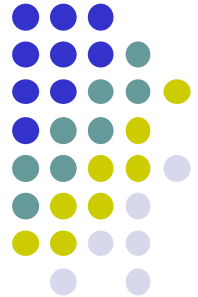
Figura 4.30 Modello di registro. A seconda della convenienza, spesso si omette di indicare i segnali R_{in} e R_{out}; in tal caso si deve assumere che essi sono sempre asseriti. Si omette pure di indicare il clock, che pure va inteso come sempre presente, a meno di avviso contrario. Il numero p di linee binarie in ingresso o in uscita è pari alla dimensione (parallelismo) del registro.

Registri a scorrimento

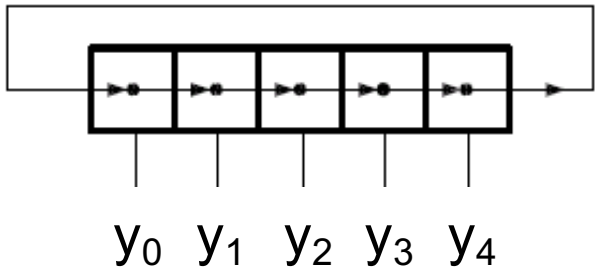


- Un registro a scorrimento prevede segnali di controllo per comandare lo scorrimento verso sinistra o destra di uno o più bit
- Se lo scorrimento è verso destra, il bit meno significativo viene perso e si antepone uno *0* come cifra più significativa (equivale ad una divisione per *2*)
- Se lo scorrimento è verso sinistra, il bit più significativo viene perso e si suppone uno *0* come cifra meno significativa (equivale ad una moltiplicazione per *2*)
- In questo caso, per evitare che il numero contenuto cambi segno, in alcuni registri a scorrimento il bit di segno (più significativo) non viene coinvolto nell'operazione

Registri ad anello



- Un registro ad anello equivale ad un registro a scorrimento in cui l'ultimo flip-flop è adiacente al primo
- Se c'è soltanto un flip-flop ad *1*, si generano *n* segnali identici con fase diversa con periodo *T=n/f* sfasati tra loro di *T/n* (*f* frequenza clock)
- A tutti gli effetti svolge la funzione di un contatore modulo *n*
- Lo useremo nella CPU



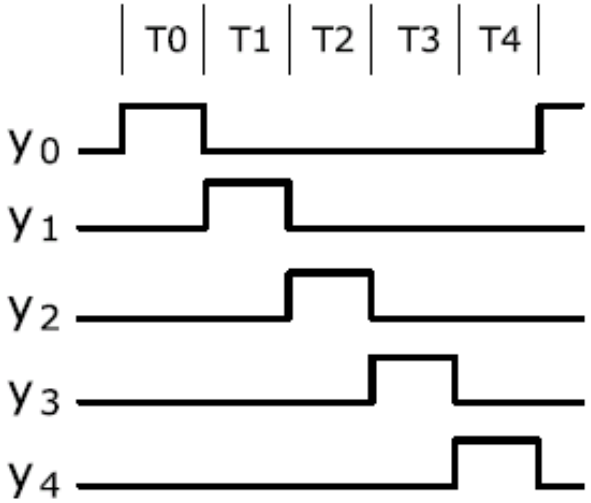
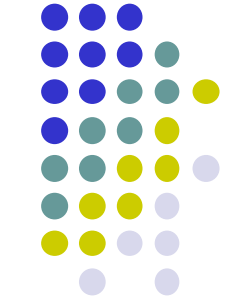


Figura 4.32 Registro ad anello (di 5 bit) e forme d'onda corrispondenti allo stato dei singoli bit. Il registro è precaricato in modo da contenere un solo 1.

Trasferimento dell'informazione



- Una larga parte dell'attività all'interno di un calcolatore consiste nel trasferimento dell'informazione tra registri
- Per trasferire l'informazione tra due registri occorre che l'uscita del registro sorgente venga portata in ingresso a quello di destinazione
- Se si hanno *m* possibili registri sorgente e *n* registri destinazione occorre una rete di interconnessione che selettivamente colleghi le sorgenti con le destinazioni
- In linea di principio si dovrebbero avere *mxn* percorsi e una logica di selezione che permetta di selezionare quelli che interessano in un determinato istante
- Vediamo una possibile soluzione tramite porte AND/OR per selezionare i registri che verranno caricati al prossimo impulso di clock

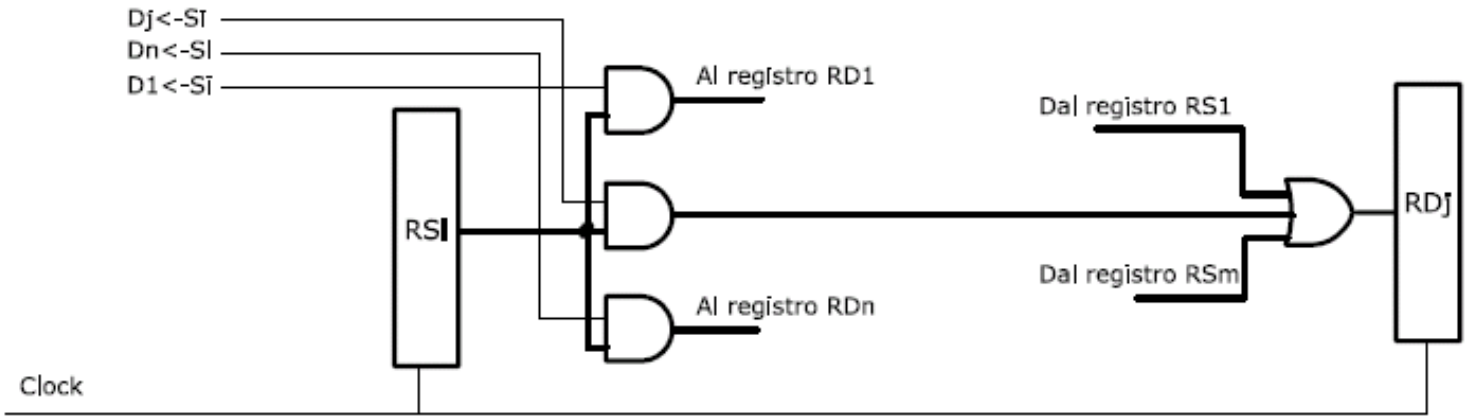
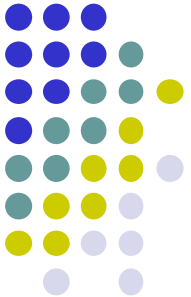


Figura 4.37 Rete di interconnessione da m registri sorgente a n registri di destinazione. I trasferimenti vengono selezionati asserendo le linee di comando $D_j \leftarrow S_i$. Si è assunto che i registri sorgente abbiano le uscite sempre disponibili (siano, cioè, senza uscita in terzo stato, oppure con uscita in terzo stato e RS_{out} sempre asserito). Si è anche assunto che il registro di destinazione abbia R_{in} sempre asserito.



- Una simile struttura permette trasferimenti multipli durante ciascun impulso di clock
- Permette anche il trasferimento da uno a più registri o il trasferimento dell'OR del contenuto di più sorgenti in una o più destinazione
- Tuttavia ha lo svantaggio della complessità e dell'occupazione di spazio sul chip o sulla piastra
- Pertanto si preferiscono le cosiddette strutture a BUS

Bus

- Vediamo un modello concettuale in cui la porta OR svolge la funzione di bus, ossia di percorso comune a tutti i trasferimenti

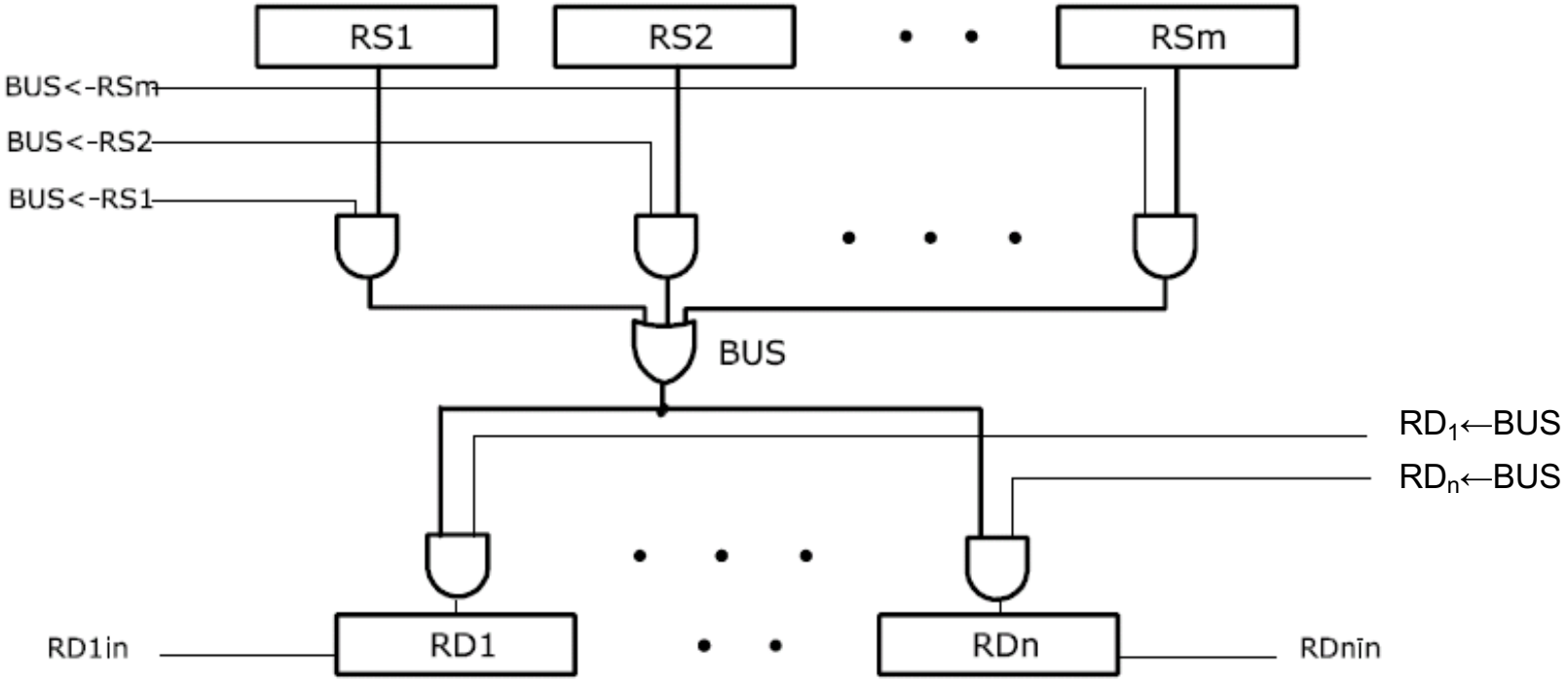


Figura 4.38 Schema concettuale della struttura a bus con normali porte AND-OR. Il trasferimento dal registro RSi al registro RSj richiede che siano asserite le due linee di comando BUS←RSi e RDj←BUS. Si è omesso di indicare il clock comune a tutti i registri.

- Lo schema appena visto ha solo valore didattico, in quanto solitamente si usa la logica con uscita a tre stati

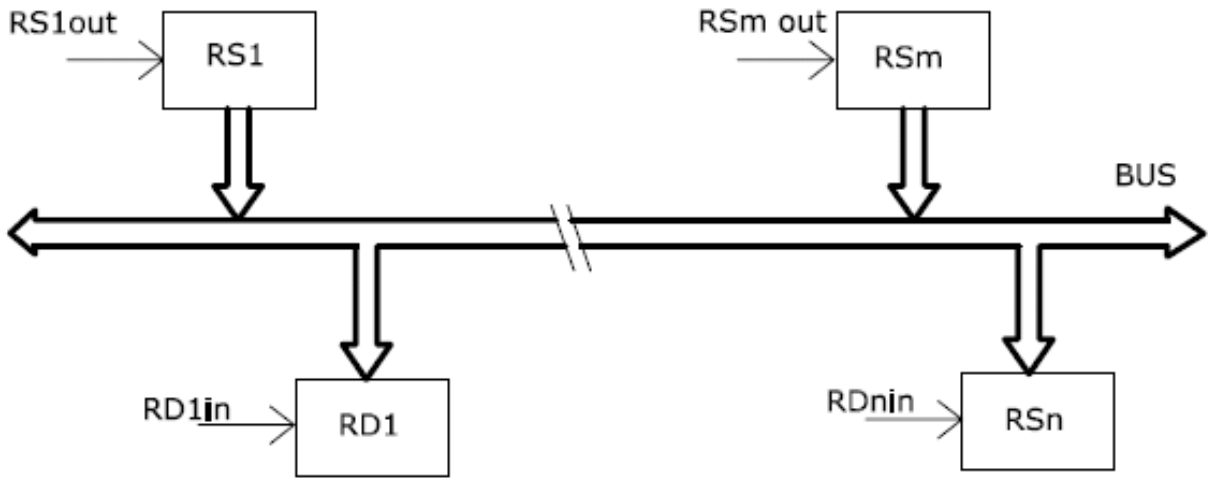
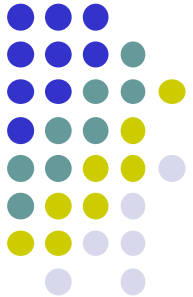


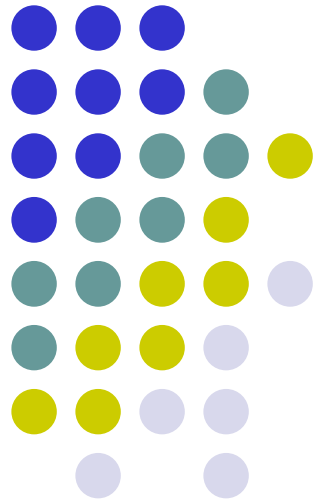
Figura 4.39 Schema di collegamento tramite bus. Si suppone le uscite dei dispositivi collegate sul bus siano a tre stati. In un dato momento deve essere asserito un solo segnale di abilitazione delle uscite, in modo che il bus venga portato allo stato logico del corrispondente registro. I segnali di abilitazione degli ingressi determinano quali registri di destinazione vengono caricati sul prossimo (fronte del) clock.



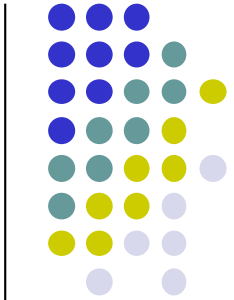
- In questo caso un solo registro può avere l'uscita abilitata
- Se tutti i registri sorgenti hanno uscita indeterminata, l'eventuale trasferimento è indeterminato
- La struttura a bus è lo standard nei calcolatori
- Ha il problema però di consentire un unico trasferimento alla volta, facendo sì che il bus diventi il "collo di bottiglia" del sistema
- Inoltre in presenza di più unità indipendenti che possono comandare l'uso del bus, si richiede una logica di arbitraggio in caso di contese per l'utilizzo

ARCHITETTURE DEGLI ELABORATORI

Prof. Massimo Tivoli



Programma del corso



- Introduzione, cenni storici e tendenze attuali
- Rappresentazione dell'informazione
- Reti logiche
- Reti sequenziali
- CPU
- Memoria
- Sottosistema di I/O
- Linguaggio macchina
- Linguaggio assembler
- Architettura x86

RIFERIMENTI

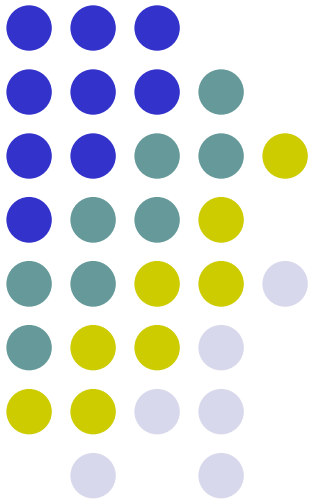
Giacomo Bucci

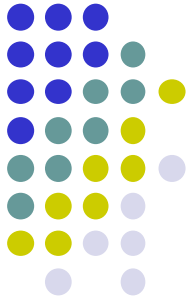
Architettura e organizzazione dei calcolatori elettronici - Fondamenti

McGraw-Hill, 2005

Il Repertorio delle Istruzioni

- Repertorio istruzioni (RISC/CISC)
- Classificazione architetture



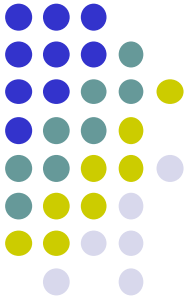


- Nel modello di Von Neumann la memoria centrale è il contenitore delle istruzioni e dei dati

- Spetta alla logica della CPU leggere le istruzioni, decodificarle e generare i comandi (segnali) che fanno eseguire le azioni previste dalla specifica istruzione

- Possiamo classificare i tipi di istruzione come segue:
  - elaborazione: istruzioni aritmetiche e logiche
  - memorizzazione: istruzioni sulla memoria
  - trasferimento: istruzioni di I/O
  - controllo: istruzioni di verifica e di salto

- L'elaborazione richiede che i dati vengano letti dalla memoria, manipolati in CPU ed eventualmente (ri)scritti in memoria



- Illustriamo ora il processo di esecuzione dei programmi, soffermandoci solo sul punto di vista del programmatore, rimandando al seguito il punto di vista architetturale ed in particolare la descrizione dell'interazione delle componenti e della logica che li comanda

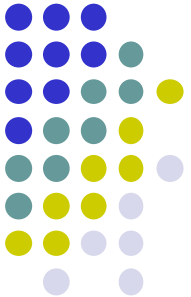
- Faremo riferimento ad un'ipotetica architettura, che verrà dettagliata nel seguito

- Introdurremo prima il repertorio delle istruzioni del linguaggio macchina, approfondendo la relazione tra la loro struttura e l'architettura

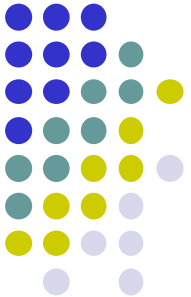
- Daremo quindi una classificazione delle architetture e approfondiremo alcuni aspetti relativi alle modalità di indirizzamento dei dati

- Iniziamo a dare uno sguardo più da vicino alla memoria (centrale), visto che in essa devono transitare sia istruzione che dati su cui operano

La memoria (centrale)

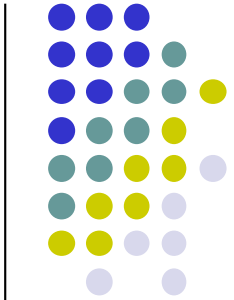


- Si compone di *celle* o *locazioni*
- Celle o locazioni:
  - composte da un numero prefissato di bit
  - accessibili tramite indirizzi (da *0* a *M-1,* dove *M* è l'estensione della memoria)
- Byte:
  - *8* bit
  - per convenzione si considera come cella elementare, per cui
    - estensione misurata in termini di byte
    - indirizzi associati ai byte



- Il byte è troppo limitato per soddisfare le esigenze di rappresentazione
- Vengono definiti quindi raggruppamenti di maggior dimensione
- Noi adotteremo le seguenti convenzioni:
  - semiparola (*half word*): coppia di byte
  - parola (*word*): gruppo ordinato di *4* byte
- Nel seguito useremo il termine *cella* o *locazione* per indicare la generica unità di memoria (byte, semiparola o parola) letta o scritta

Operazioni di memoria



- Lettura *(load):*
  - vengono trasmessi alla memoria l'indirizzo della cella da leggere e il comando di lettura
  - la memoria restituisce il contenuto della cella indirizzata
- Scrittura *(store):*
  - vengono trasmessi alla memoria l'indirizzo della cella da leggere, il dato da scrivere e il comando di scrittura
  - la memoria scrive il dato nella cella indirizzata
- Letture e scritture solitamente non vengono effettuate a byte, ma a parole
- Infatti il maggior parallelismo causa un aumento delle prestazioni dovuto alla riduzione del numero di potenziali trasferimenti
- Vediamo una figura che schematizza una memora a *32* bit di *M* byte

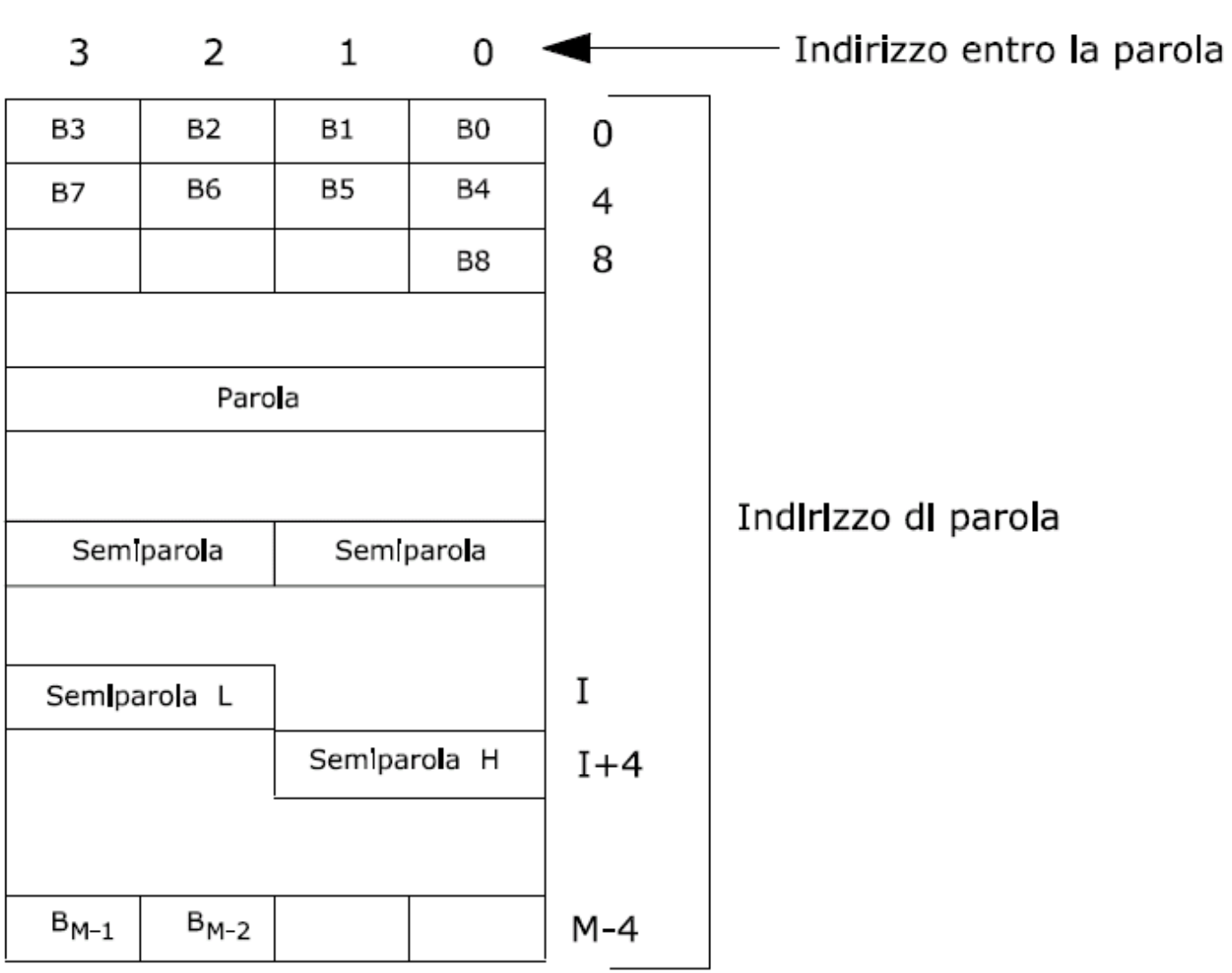
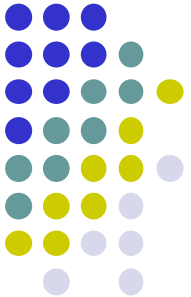
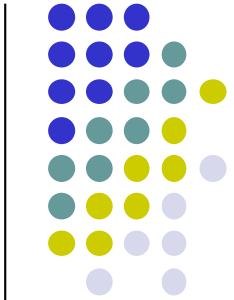


Figura 5.1 Schematizzazione di una memoria a 32 bit di M byte. A destra viene riportato l'indirizzo del byte meno significativo della corrispondente parola. Questo può essere preso come l'indirizzo di parola. L'indirizzo di un generico byte è ottenuto sommando l'indirizzo di parola con l'indirizzo entro la parola.



- Si noti che le parole possono essere non *“allineate”* (indirizzo non multiplo di *4*)
- Come vedremo ciò pone complicazioni nelle operazioni di lettura/scrittura
- Nel seguito assumeremo sempre di andare ad effettuare letture/scritture di parole allineate

La codifica delle istruzioni



- Le istruzioni del linguaggio sono costituite da sequenze di bit e si dividono in due parti:
  - *codice operativo:* è sempre presente e serve per specificare l'operazione da compiere
  - parte *operandi:* serve per specificare una o più dati a cui l'istruzione fa riferimento
- Assumiamo che ogni istruzione occupi esattamente una parola
- Inoltre supponiamo che la *CPU* disponga di un numero di registri di uso generale denominati *R0, R1, R2,….* ciascuno di *32* bit
- Vediamo due possibile formati per la codifica delle istruzioni

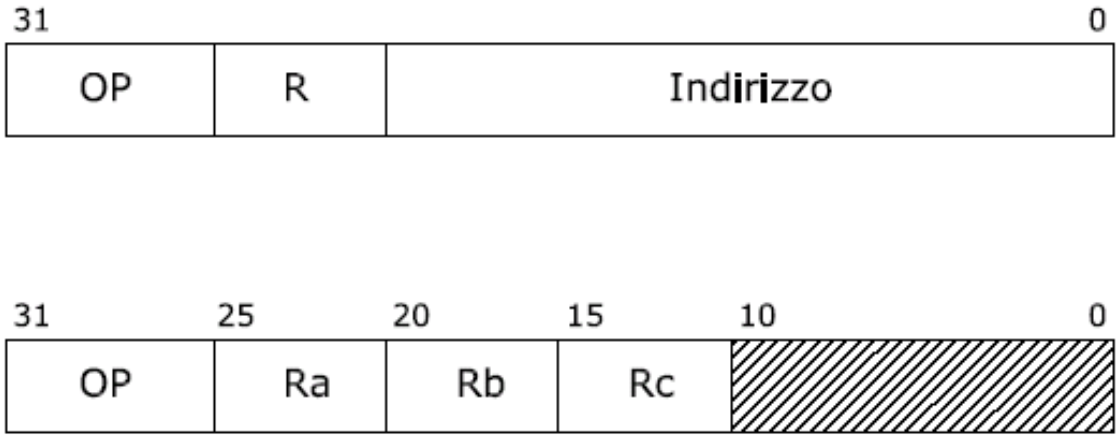
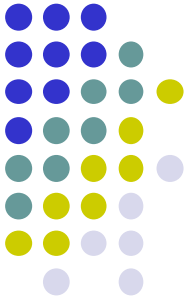
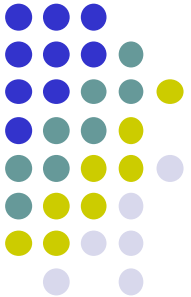


Figura 5.2 Due possibili formati per la codifica delle istruzioni di macchina su parole di 32 bit. Il campo OP contiene il codice di operazione ed è sempre presente; gli altri campi dipendono dal codice di operazione stesso. Il primo formato prevede due operandi: un registro di CPU e un indirizzo di memoria. Il secondo formato prevede tre operandi, tutti registri di CPU. Il campo tratteggiato è inutilizzato. La numerazione dei bit all'interno delle parole adotta la convenzione di assegnare il numero 0 al meno significativo.



- La parola è suddivisa in più campi
- I campi sono diversi, nel formato e nel significato, a seconda dell'istruzione
- Il campo OP corrispondente al codice operativo è sempre presente e per semplicità assumeremo che occupi sempre i *6* bit più significativi
- Poiché il codice operativo codifica le diverse istruzioni, è possibile avere al più $2^6=64$ istruzioni diverse
- Inoltre poiché i campi che specificano registri sono di *5* bit, è possibile avere al più $2^5=32$ registri di uso generale



- Al primo formato in Figura 5.2 può ad esempio corrispondere ad un'istruzione che simbolicamente scriviamo come

  LD R1, Var

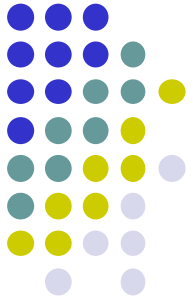
- Essa consiste nel caricamento (*LD, Load*) in un registro (*R1*) del contenuto della cella di memoria il cui indirizzo è specificato nell'operando (*Var*)

- *LD* è il *codice mnemonico* per *Load*, *R1* è l'identificatore del registro di destinazione, *Var* è il nome simbolico di una posizione di memoria (variabile)

- In forma compatta l'effetto di tale istruzione può essere descritto con

  R1 ← M[Var]

  dove *M[Var]* rappresenta il contenuto della cella di memoria di nome *Var*, l'operatore "←" assegna al termine di sinistra il valore del termine di destra

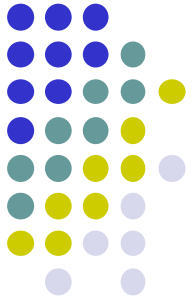


- Supponiamo che
  - l'operazione LD è codificata con la sequenza di bit *110011*
  - come già accennato il campo che identifica il registro occupa *5* bit (ovvero ci sono $2^5=32$ registri)
  - al simbolo *Var* corrisponde, ad esempio, l'indirizzo *1023*

- Allora l'istruzione *LD R1, Var* si codifica come

  110011 00001 00000000001111111111

- Chiaramente si preferisce riferirsi alle istruzioni con la rappresentazione simbolica anziché numerica



- Al secondo formato in Figura 5.2 può ad esempio corrispondere l'istruzione a tre operandi

  ADD R1, R2, R3 ;R1 ← R2 + R3

- Essa consiste nella somma (*ADD*) del contenuto di due registri (*R2 e R3*) e memorizzazione del risultato in un terzo registro (*R1*)

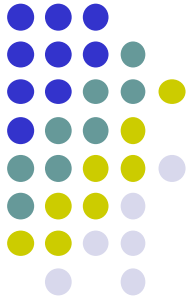
- *ADD* è il *codice mnemonico*, *R1* è l'identificatore registro di destinazione, *R2* e *R3* sono gli identificatori dei registri sorgenti

- L'effetto in forma compatta dell'istruzione è stato riportato come commento dopo il ";"

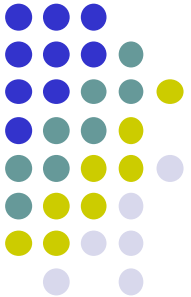
- Si noti che gli *11* bit meno significativi non vengono utilizzati

- D'ora in poi per le istruzioni simboliche useremo la seguente sintassi:
  - codice mnemonico
  - primo operando: destinazione (in alcuni casi anche sorgente)
  - secondo operando: sorgente
  - terzo operando: (se esiste) sorgente

Programmi e processi di esecuzione



- Un programma è una sequenza di istruzioni (statement)
- E' la traduzione in un linguaggio di programmazione di un algoritmo, ossia di un procedimento passo-passo che risolve un problema dato
- Un calcolatore è in grado di interpretare ed eseguire istruzioni molto semplici (come le precedenti) dette istruzioni di macchina
- Una sequenza di istruzioni di macchina costituisce un programma in linguaggio macchina
- La corrispondente sequenza in forma simbolica costituisce un programma in linguaggio assembler (o assembly)
- L'assembler libera il programmatore dalla necessità di comunicare in forma binaria associando nomi simbolici a istruzioni (codici mnemonici) e a indirizzi di memoria (etichette e variabili)
- La traduzione da un programma scritto in linguaggio assembler all'equivalente in linguaggio macchina è un procedimento meccanico e può essere delegato ad un altro programma, scritto in linguaggio macchina
- Tale programma è chiamato assemblatore



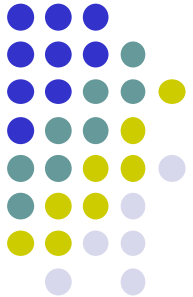
- La sintassi di un'istruzione assembler è la seguente:

  LABEL OP OPN1, OPN2, … ; Commento

  - *Label* è un'etichetta simbolica, opzionale, assegnata allo statement
  - *OP* è il codice mnemonico dell'operazione da eseguire
  - *OPNi* sono gli eventuali parametri (da zero a più a seconda di *OP*)
  - *Commento* è un libero commento di solito preceduto da ";"

- Il linguaggio assembler non aggiunge potenza descrittiva al linguaggio macchina, poiché le istruzioni corrispondono a una o a un numero limitato di istruzioni di un linguaggio macchina

- Inoltre, anche se fa utilizzato di nomi simbolici, è fortemente dipendente dalla particolare architettura



- Sorse così l'esigenza di avere linguaggi di programmazione ad alto livello, ossia aventi le seguenti caratteristiche:
  - indipendenti dall'architettura, in modo da rendere possibile la *portabilità* dei programmi tra architetture diverse
  - istruzioni più sintetiche, più potenti, più vicine al modo di ragionare umano

- Compilatore: traduce un programma in un linguaggio ad alto livello in uno equivalente in linguaggio macchina per permettere la sua esecuzione da parte dell'elaboratore

- Interprete: simile al compilatore, ma esegue la traduzione in fase d'esecuzione, istruzione per istruzione

- Assemblatori, compilatori e interpreti vengono chiamati genericamente *traduttori*

- Alcuni linguaggi ad alto livello (in ordine cronologico): Fortran, Cobol, Lisp, Basic, Pascal, C, C++, Java, …)

- La seguente figura schematizza il processo di scrittura-traduzione-esecuzione di programmi scritti in linguaggi ad alto livello

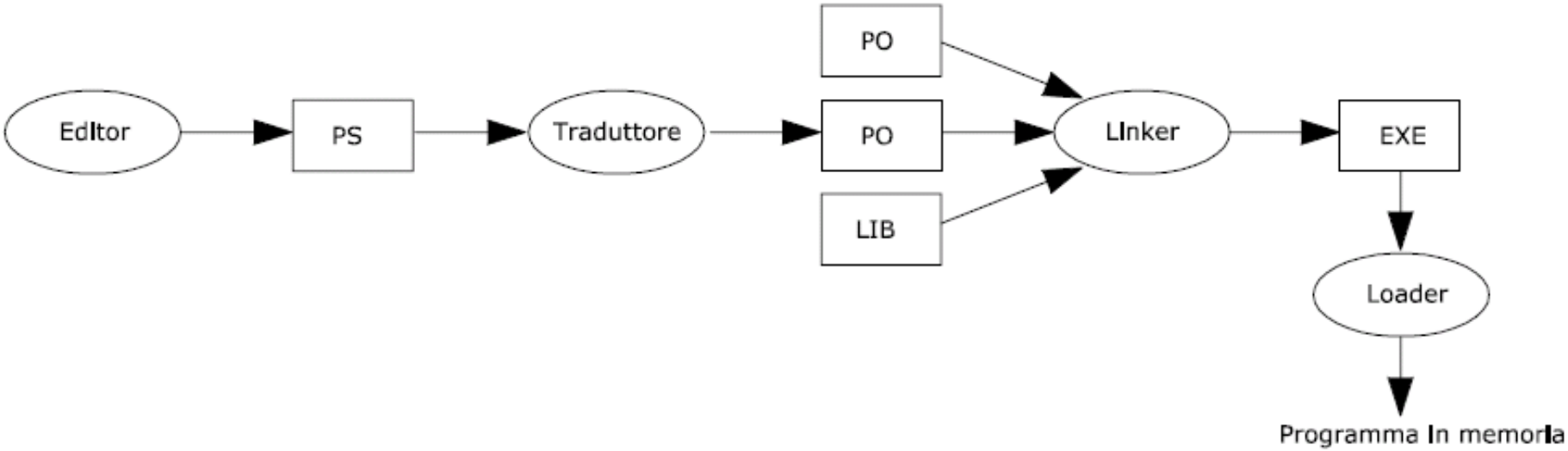
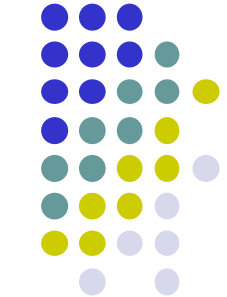


Figura 5.3 Il processo Scrittura-Compilazione-Esecuzione.

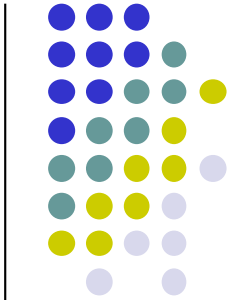
Istruzioni e architettura



- Vediamo le conseguenze delle scelte architetturali sul repertorio delle istruzioni
- Si consideri lo statement in un linguaggio ad alto livello (C) in cui *a,b* e *c* sono tre variabili intere

  a=b+c;

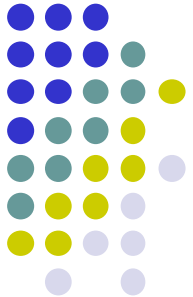
- Supponiamo che le tre variabili corrispondano alle locazioni che simbolicamente indichiamo con *A,B* e *C*
- Essa richiede:
  1. lettura della parola all'indirizzo *B*
  2. lettura della parola all'indirizzo *C*
  3. somma tramite ALU
  4. scrittura del risultato nella parola di indirizzo *A*
- Vediamo quattro possibili soluzioni



Soluzione 1: modello *memoria-memoria*

ADD A, B, C

- I passi da 1. a 4. vengono cioè eseguiti da una singola istruzione
- Svantaggi:
  - complicazione della logica di esecuzione, poiché i passi da 1. a 4. devono essere sequenzializzati (una sola operazione di memoria alla volta)
  - formato molto ampio dovuto a tre campi indirizzo
- Le istruzioni possono specificare tre indirizzi
- Macchina a tre indirizzi

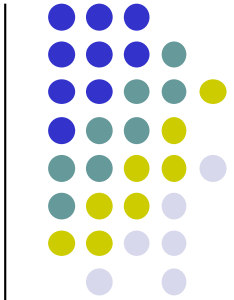


Soluzione 2: modello *registro-registro*

```
LD    R1,B
LD    R2,C
ADD   R3,R1,R2
ST    A,R3
```

- L'istruzione *ST* (store) è l'inversa della *LD* (load), ossia ha l'effetto di memorizzare il contenuto del registro (*R3*) nella corrispondente cella di memoria (*A*)
- Solo le istruzioni *LD* e *ST* specificano un indirizzo
- Logica di esecuzione molto semplice, poiché le altre istruzioni manipolano solo dati già presenti in CPU

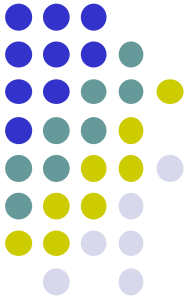
Soluzione 3: modello *registro-memoria*



```
LD    R1,B
ADD   R1,C
ST    A,R1
```

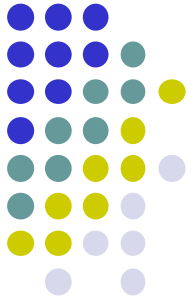
- Soluzione intermedia tra la 1 e la 2, che come vedremo sembra essere la prevalente
- Nelle prime versioni i registri generali non erano equivalenti, ma uno particolare, detto *accumulatore*, veniva implicitamente designato come registro destinazione
- Oltre a *LD* e *ST*, anche le altre istruzioni (come *ADD*) possono specificare un indirizzo
- Macchina ad un indirizzo

Soluzione 4: modello *stack*



```
PUSH    B
PUSH    C
ADD
POP     A
```

- Lo stack è una struttura dati LIFO (Last In First Out): l'ultimo dato inserito con una PUSH è il primo ad essere prelevato con una *POP*
- La *ADD* esegue prima due *POP* dallo stack per prelevare i due operandi e poi una *PUSH* per inserirvi il risultato
- Le istruzioni non contengono indirizzi
- Macchina a 0 indirizzi



Riassumendo

- in base al modello di esecuzione, ovvero al modo in cui la CPU accede e manipola gli oggetti dell'elaborazione, abbiamo definito quattro modelli (stack, registro-registro, registro-memoria e memoria-memoria)

- li abbiamo confrontati in riferimento all'istruzione di alto livello *a=b+c:*

stack	*registro-registro*	*registro-memoria*	*memoria-memoria*
PUSH B *PUSH C* *ADD* *POP A*	*LD R1,B* *LD R2,C* *ADD R3,R1,R2* *ST A,R3*	*LD R1,B* *ADD R1,C* *ST A,R1*	*ADD A, B, C*
0 indirizzi	Load/Store	*1* indirizzo	*3* indirizzi

- I modelli prevalenti sono registro-memoria e registro-registro, che riducono gli accessi a memoria