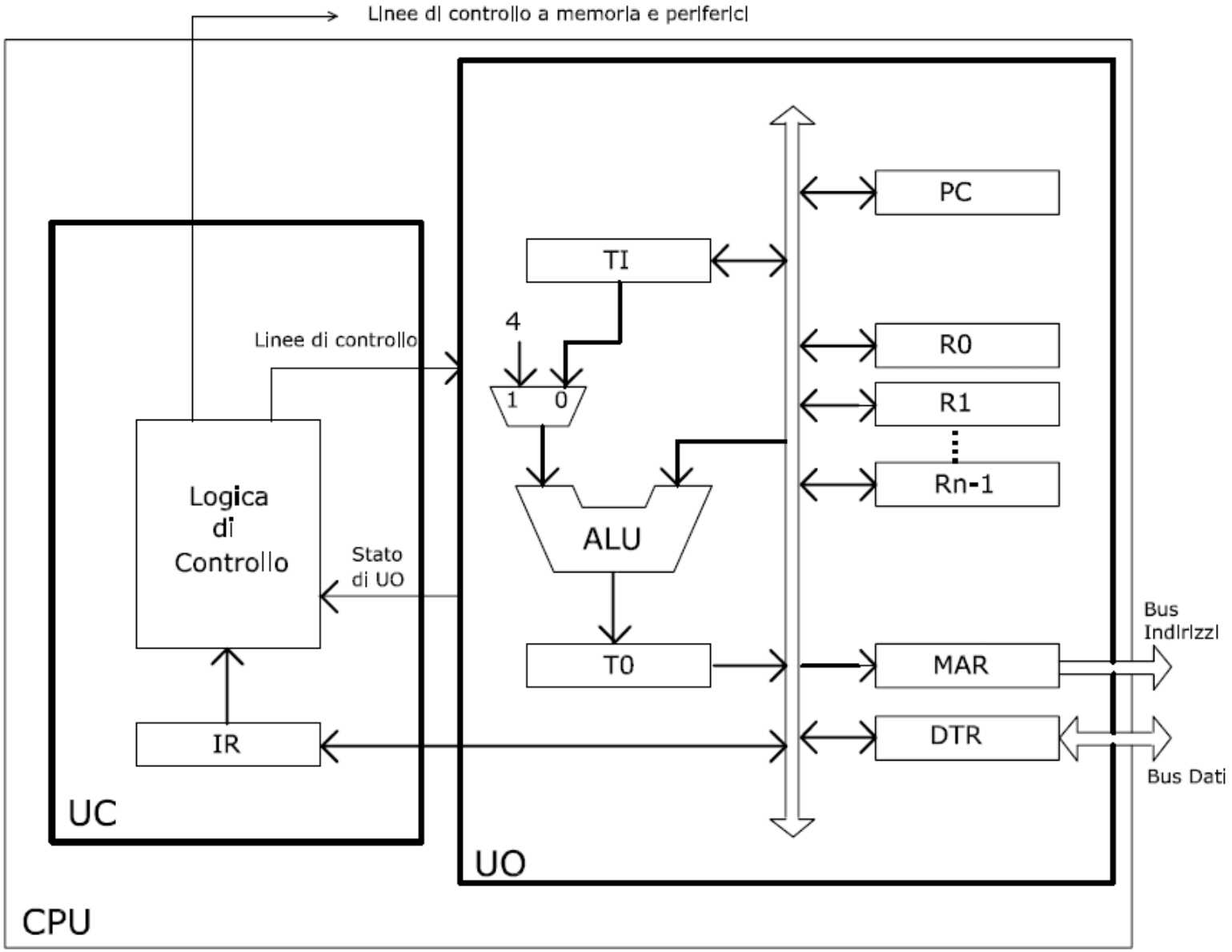


Figura 6.6 Organizzazione del percorso dati a singolo bus interno.

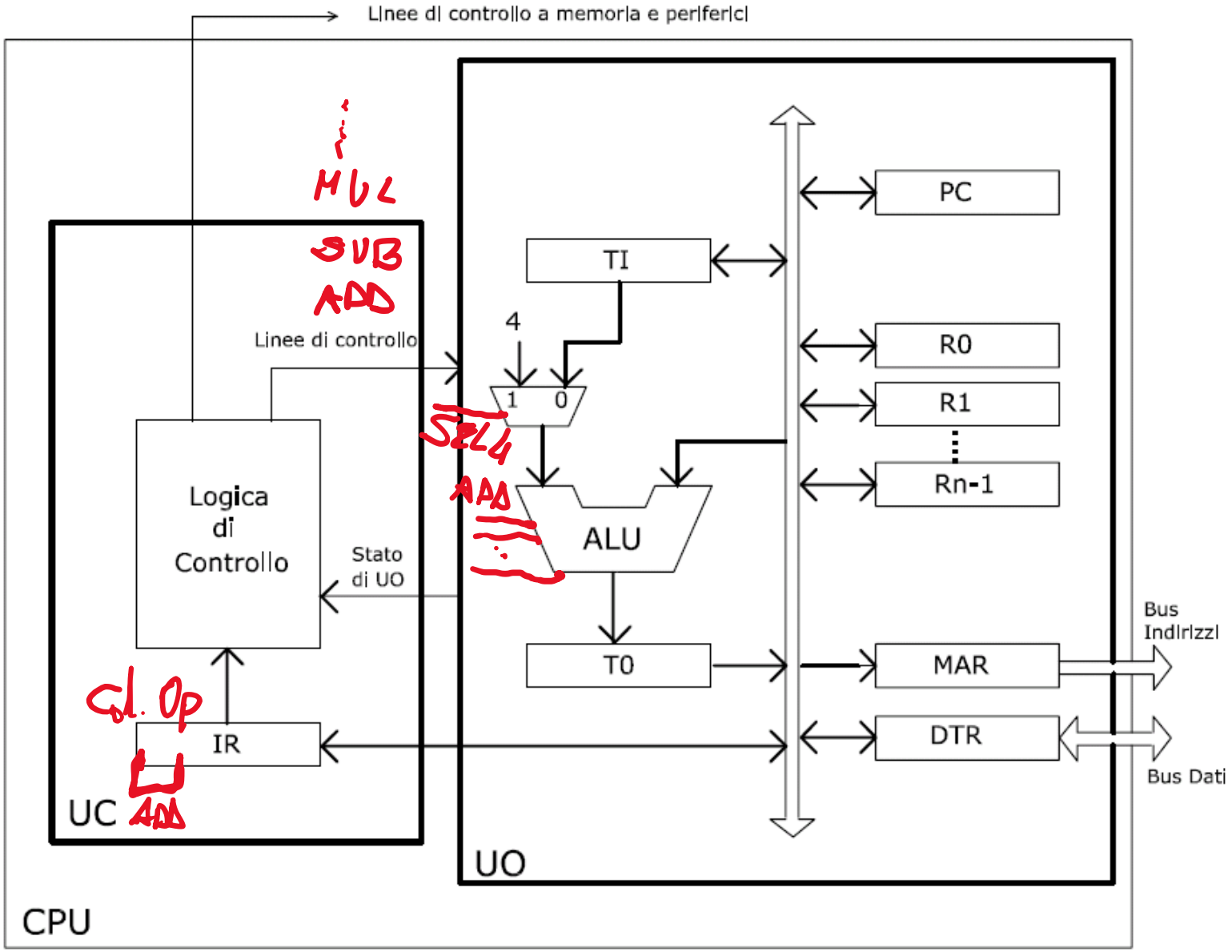


Figura 6.6 Organizzazione del percorso dati a singolo bus interno.

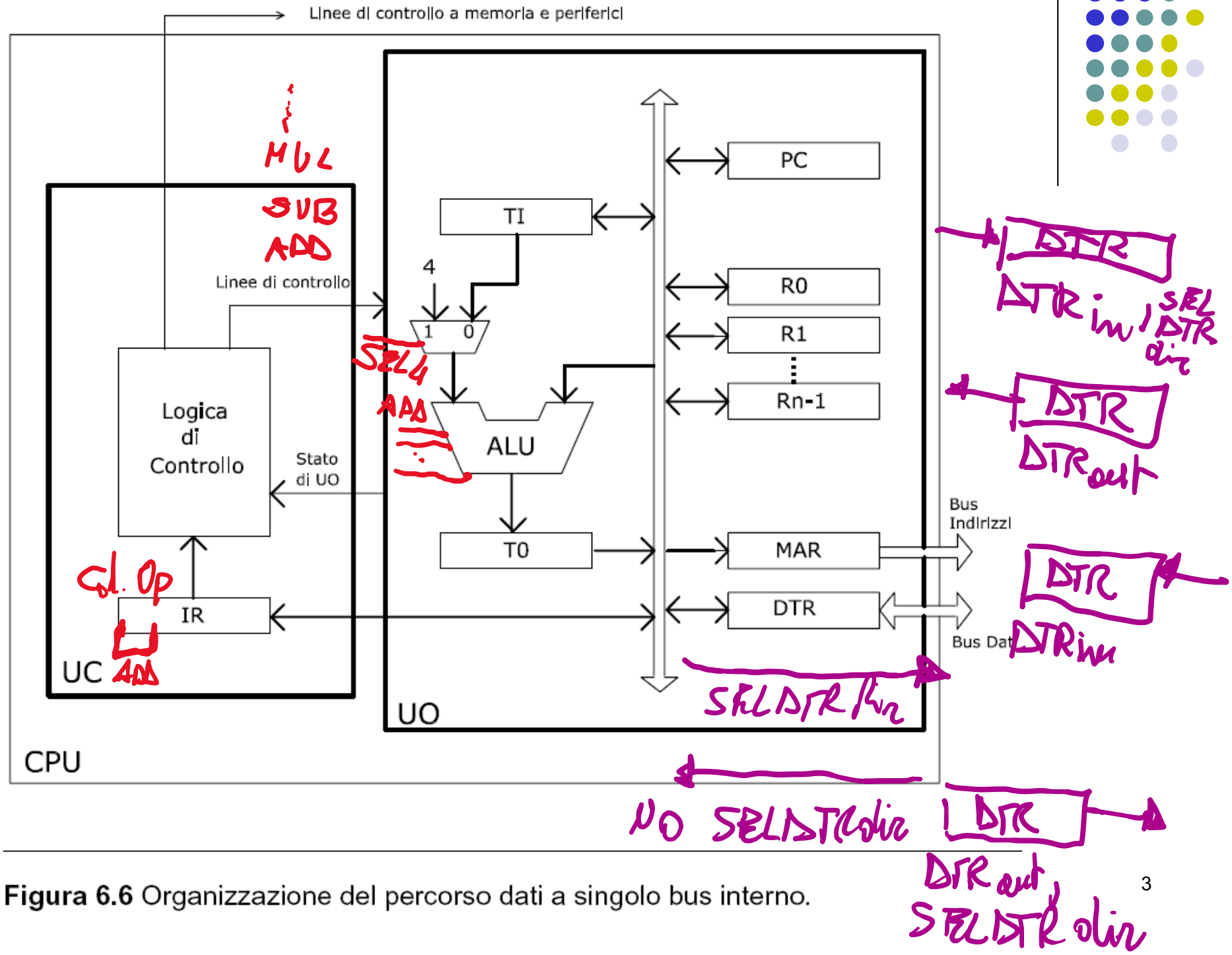
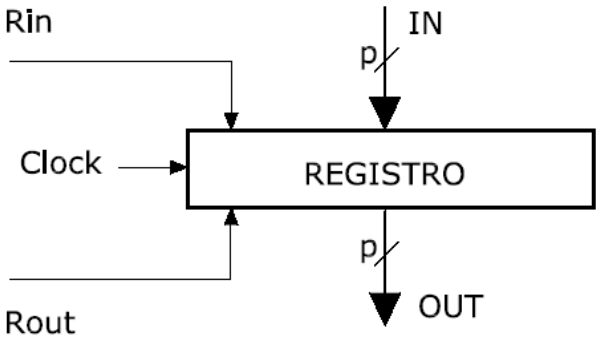
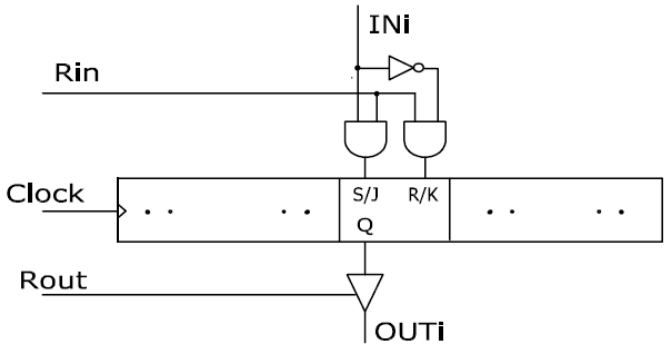


Figura 6.6 Organizzazione del percorso dati a singolo bus interno.

Trasferimento registro-registro

- Ricordiamo brevemente il funzionamento dei registri:





- Per scrivere nel registro R_{in} deve essere posto ad 1; la scrittura termina subito dopo il fronte di discesa del clock

- Per poter leggere il contenuto, R_{out} deve essere posto ad 1

- La figura seguente mostra il collegamento di un registro al bus interno e la temporizzazione di una scrittura da un registro *RB* al registro *RA (RA ← RB)*

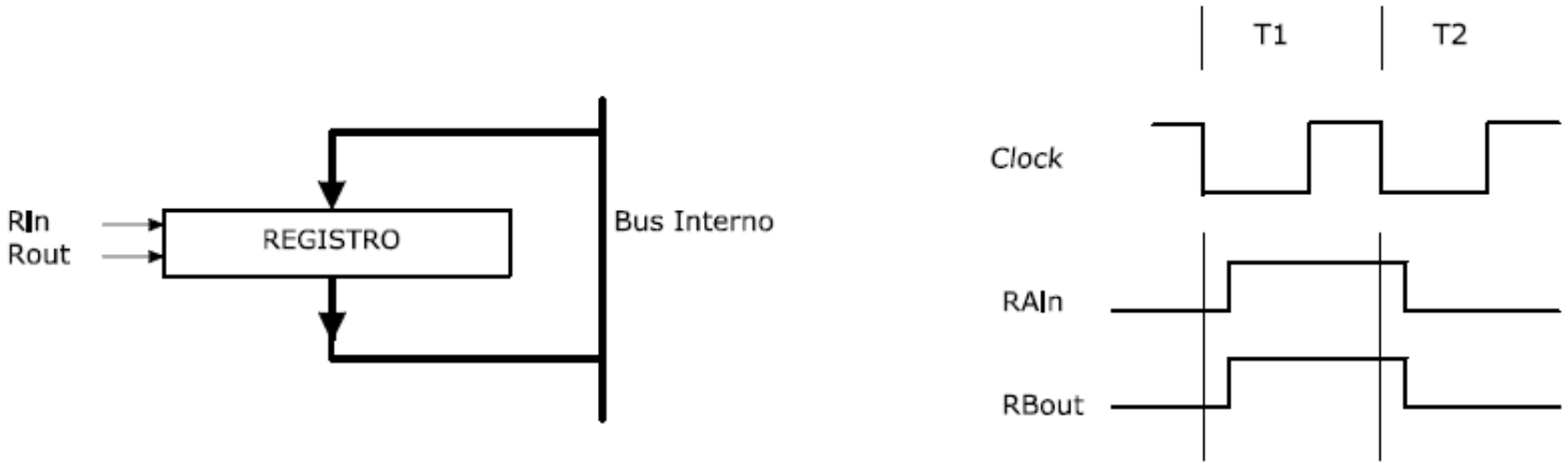
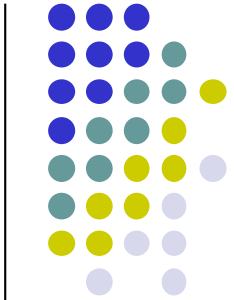


Figura 6.7 Collegamento di un registro al bus interno e temporizzazione di un trasferimento tra registri.

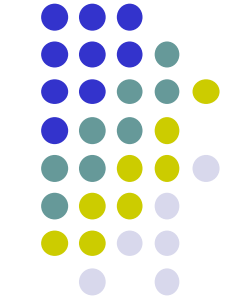


- In pratica i segnali RB_{out} e RA_{in} vengono asseriti all'inizio del clock T1
- Sul fronte di discesa di T1 il registro *RA* memorizza il dato in ingresso
- Tale comportamento può essere descritto in forma compatta nel seguente modo:

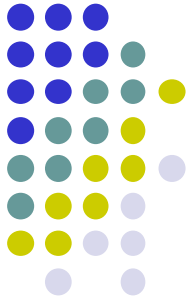
  $T1: \ RB_{out}, RA_{in}$

  ad indicare che RB_{out} e RA_{in} vengono asseriti durante T1

Impiego dell'ALU



- Supponiamo che l'ALU abbia la stessa struttura semplificata studiata precedentemente
- La presenza di un solo bus interno, vincolando ad un trasferimento per volta, impone l'uso di due registri di appoggio per l'ALU:
  - *TI:* per appoggiare uno dei due operandi
  - *TO:* per il risultato
- *TI* viene presentato all'ALU tramite un selettore il cui secondo ingresso è la costante (codificata in binario) *4* (vedremo fra poco il perché)
- La linea di comando *SEL4* del selettore, se asserita, presenta in ingresso *4*, altrimenti il contenuto di *TI*
- Vediamo quindi i passi necessari per effettuare la somma di due registri, *RA* e *RB,* nel registro *RD (RD ← RA+RB)*

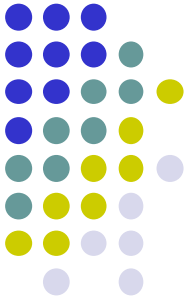


- Sono necessari tre passi:
  1. si porta il contenuto di *RA* in *TI*
  2. tramite il bus si porta il contenuto di *RB* all'altro ingresso dell'ALU e si comanda all'ALU la somma, depositando il risultato in *TO*
  3. *TO* viene copiato in *RD*

- L'UC deve quindi generare i seguenti segnali di controllo:

  T1: RA_{out} , TI_{in}
  T2: RB_{out} , *ADD*, TO_{in}
  T3: TO_{out} , RD_{in}

Il registro *DTR*



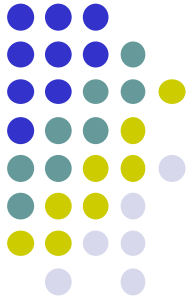
- Abbiamo già visto che il registro *DTR* è bidirezionale
- Per stabilire il senso è necessario aggiungere un altro segnale di controllo, *SELDTRdir*, che quando non asserito trasferisce dall'esterno all'interno, altrimenti dall'interno all'esterno
- La lettura di *DTR* (verso l'interno) richiede

  T1: DTR_{out}

- La scrittura in *DTR* (verso l'esterno) richiede

  T1: $SELDTRdir, DTR_{in}$

Aggiornamento del *PC*



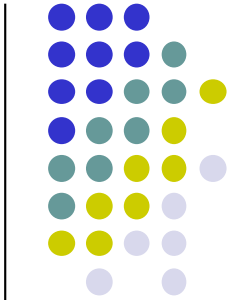
- Assumendo che le istruzioni occupino *4* byte, ad ogni ciclo di fetch il contenuto del *PC* deve essere incrementato di *4*

- Ecco perché il selettore di ingresso dell'ALU sceglie tra la costante *4* e *TI*

- L'incremento di *PC* richiede due passi:

  T1: PC_{out} , *SEL4, ADD,* TO_{in}
  T2: TO_{out} , PC_{in}

Fase di fetch



- Inizia con il prelievo dell'istruzione in memoria e si conclude nella sua scrittura nel registro *IR*

- Procede seguendo i seguenti passi:
  1. il contenuto di *PC* viene trasferito in *MAR*
  2. l'uscita di *MAR* viene presentata sul bus indirizzi e viene asserito il comando di lettura della memoria; il contenuto della cella indirizzata viene caricato in *DTR*
  3. *DTR* viene copiato in *IR*

- Possiamo determinare così i seguenti macrostati

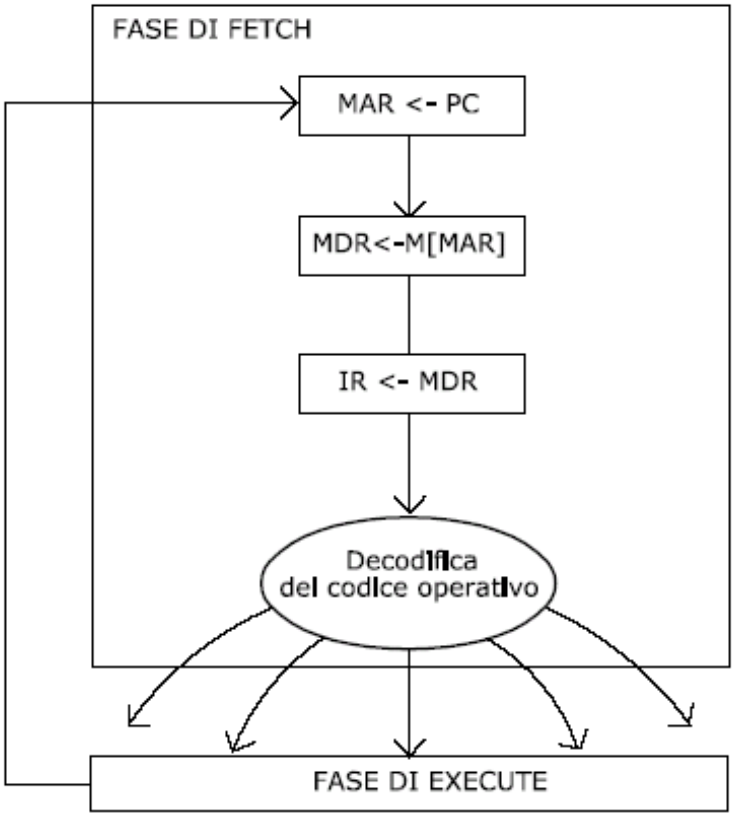
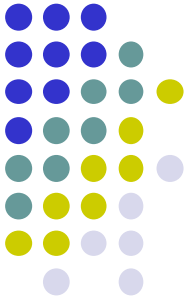
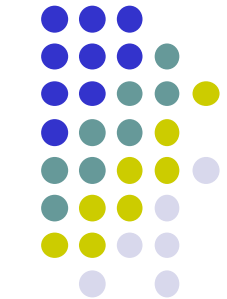


Figura 6.8 Diagramma a stati (aggregati) relativo alla fase di fetch. Il ramo di uscita dalla fase di fetch è selezionato in base al codice di istruzione.



- Assumendo che la memoria risponde entro il clock successivo, e che in tutto il tempo il comando *MRD* (Memory ReaD) debba essere asserito, abbiamo quindi la seguente sequenza di segnali di controllo:

  T1: PC_{out} , MAR_{in}
  T2: *MRD*
  T3: *MRD*, DTR_{in}
  T4: DTR_{out} , IR_{in}



- Una volta eseguiti i passi precedenti, bisogna infine aggiornare il contenuto del *PC* per predisporre la fase di fetch successiva:

  T5: PC_{out}, *SEL4, ADD,* TO_{in}
  T6: TO_{out}, PC_{in}

- Quindi in totale la fase di fetch richiede *6* cicli di clock

- Tenendo conto delle limitazioni imposte dalla struttura a singolo bus interno, osserviamo che durante il ciclo T1 il contenuto di *PC* è presente nel bus

- Quindi, senza interferire con le altre operazioni svolte durante T1, possiamo approfittare per far entrare *4* all'altro ingresso dell'ALU ed effettuare la somma

$T1: PC_{out}, MAR_{in}$

$T2: MRD$

$T3: MRD, DTR_{in}$

$T4: DTR_{out}, IR_{in}$

$T5: PC_{out}, SEL4, ADD, TO_{in}$

$T6: TO_{out}, PC_{in}$

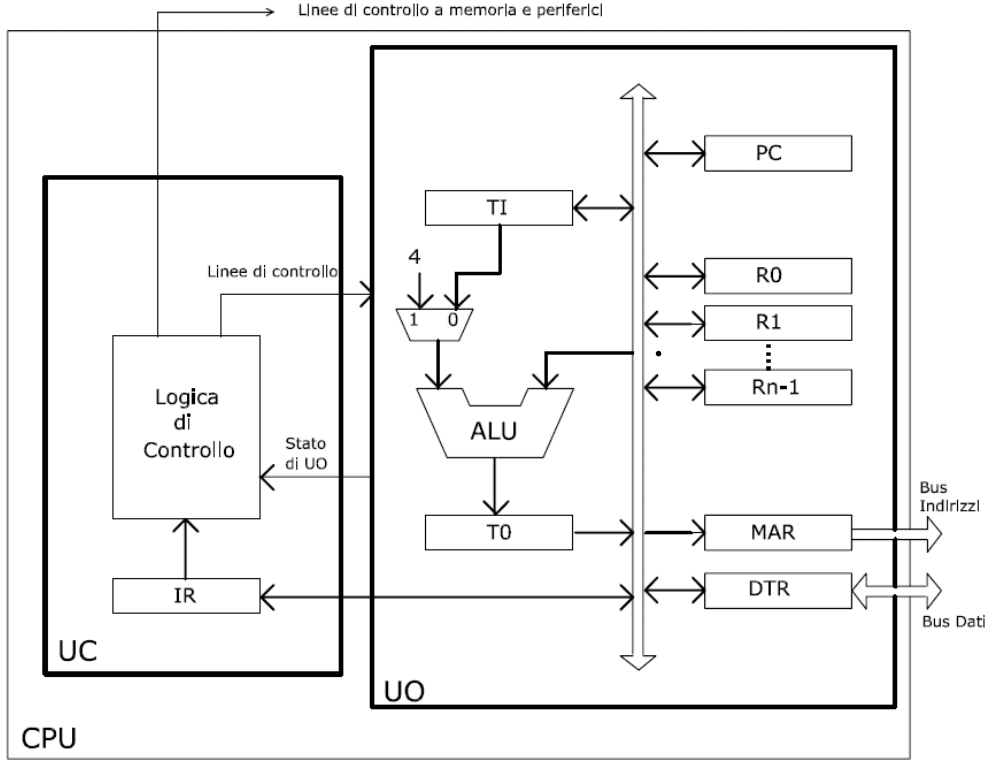
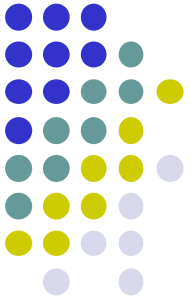


Figura 6.6 Organizzazione del percorso dati a singolo bus interno.



- In sostanza, è possibile rendere più efficiente la fase di fetch, riducendola a *4* cicli di clock:

  T1: PC_{out} , MAR_{in} , *SEL4, ADD, TOin*
  T2: *MRD,* TO_{out}, PC_{in}
  T3: *MRD,* DTR_{in}
  T4: DTR_{out} , IR_{in}

- Al termine del ciclo T4 l'istruzione letta sarà contenuta in *IR*

- La figura seguente mostra la temporizzazione dei segnali di controllo della fase di fetch nella realizzazione appena descritta

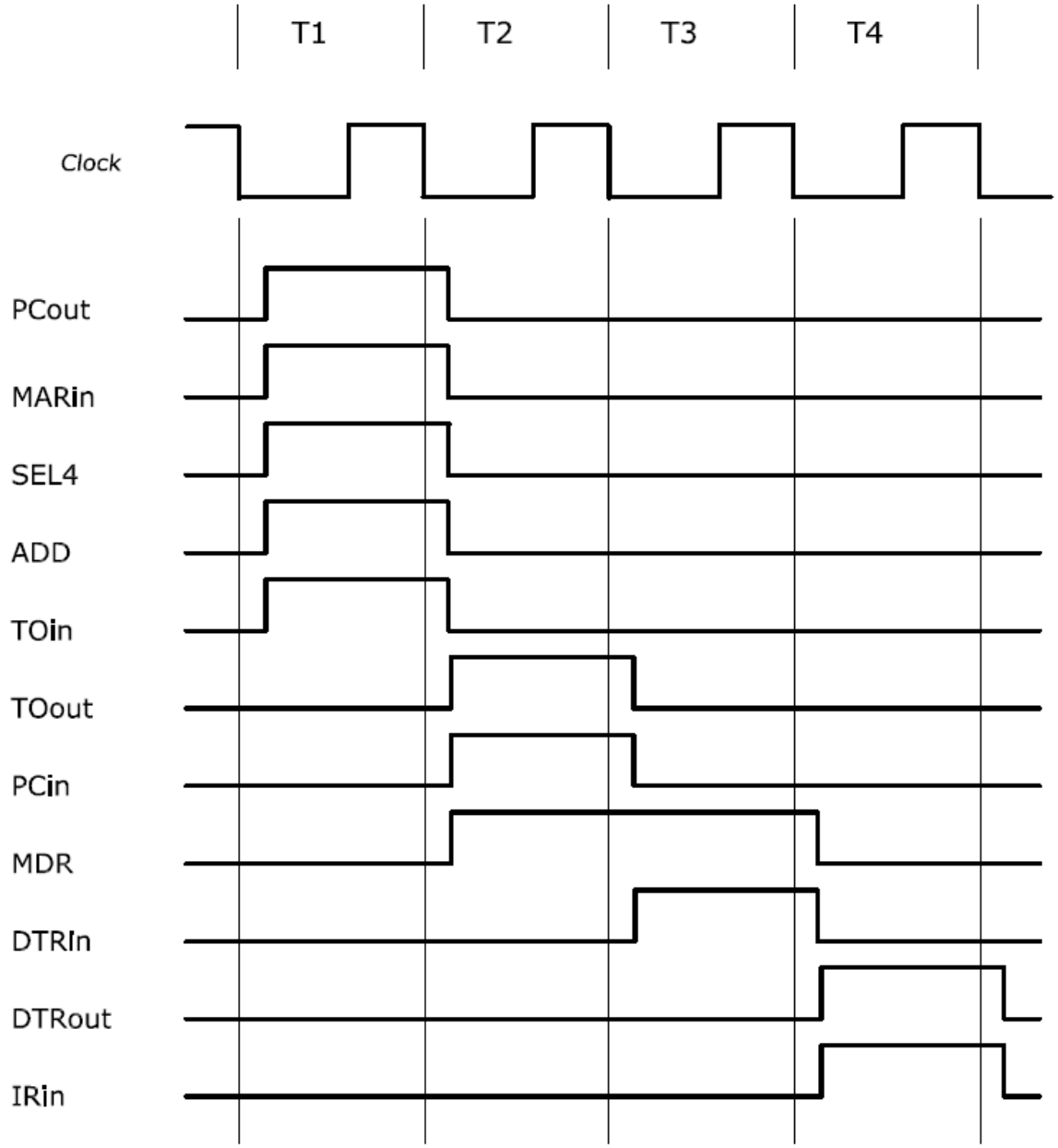
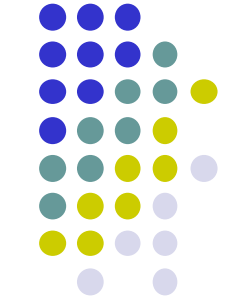


Figura 6.9 Tempificazione della fase di fetch per la CPU di Figura 6.6.



- Nel caso in cui la lettura richiedesse più di due cicli, si possono adottare soluzioni appropriate tipo quella descritta con il segnale di *WAIT*

- Quando asserito, fa rimanere la fase di fetch nel corrispondente macrostato:

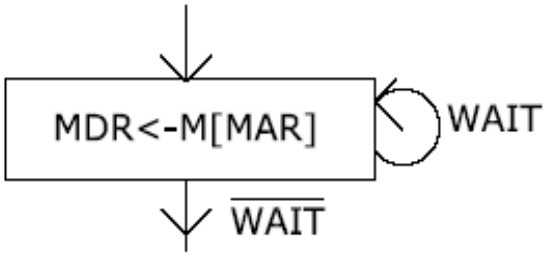


Figura 6.10 Modifica del diagramma di stato per controllare la linea di WAIT.

Fase di esecuzione

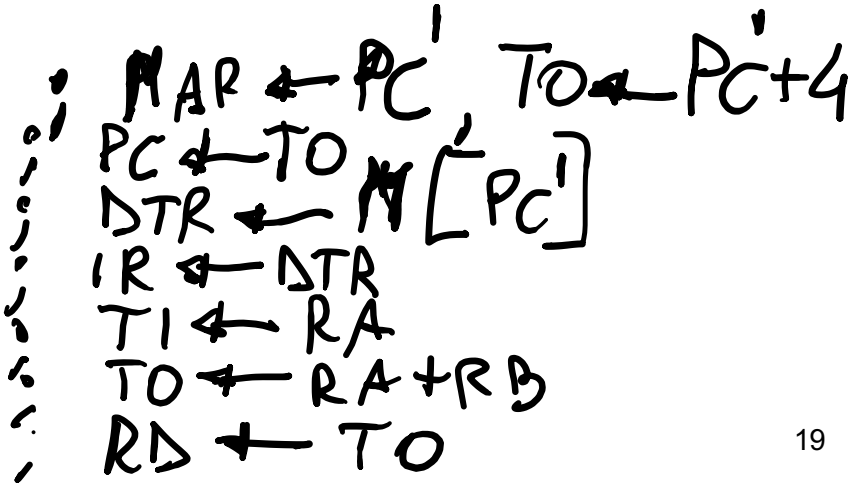
- Vediamo ora alcuni esempi di fasi di execute

- Operazioni aritmetiche: *ADD RD, RA, RB* ; *RD ← RA+RB*

  T5: RA_{out} , TI_{in}
  T6: RB_{out} , *ADD,* TO_{in}
  T7: TO_{out} , RD_{in}

  Includendo la fase di fetch :

  T1: PC_{out} , MAR_{in} , *SEL4, ADD, TOin*
  T2: *MDR,* TO_{out}, PC_{in}
  T3: *MDR,* DTR_{in}
  T4: DTR_{out} , IR_{in}
  T5: RA_{out} , TI_{in}
  T6: RB_{out} , *ADD,* TO_{in}
  T7: TO_{out} , RD_{in}



- Operazioni con la memoria: *ST V(RB),RA* *M[V+RB] ← RA*

Assumendo che

- *V* sia codificato nei 16 bit meno significativi di *IR (IR[15:0])*
- $0^{16}||IR[15:0]$ sia l'indirizzo di 32 bit ottenuto anteponendo 16 bit nulli a *IR[15:0]*
- la scrittura in memoria richieda due cicli di clock

T5: RB_{out} , TI_{in}
T6: $0^{16}||IR[15:0]_{out}$, *ADD,* TO_{in}
T7: TO_{out} , MAR_{in}
T8: *SELDTRdir,* RA_{out} , DTR_{in}
T9: *SELDTRdir,* DTR_{out} , *MWR*
T10: *SELDTRdir,* DTR_{out} , *MWR*

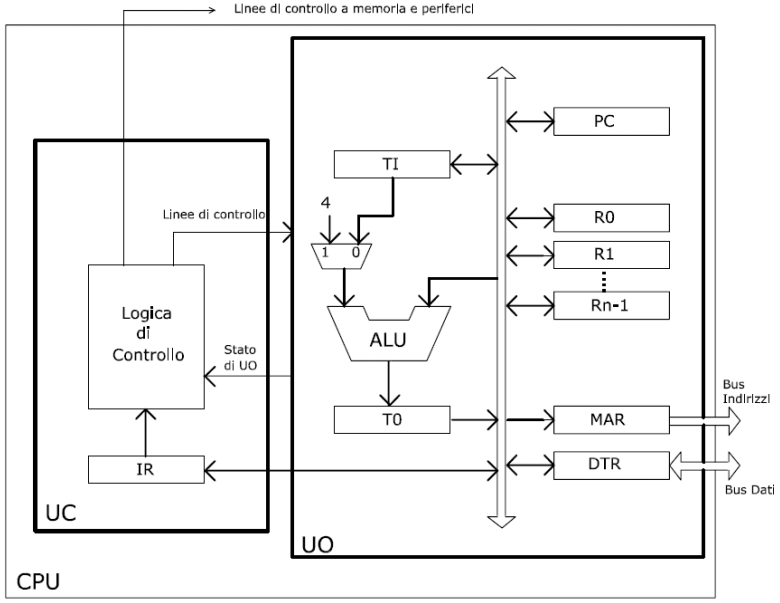
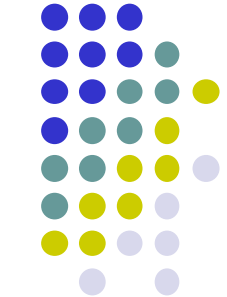


Figura 6.6 Organizzazione del percorso dati a singolo bus interno.

Architettura a più bus interni



- Le prestazioni della CPU, chiaramente dipendono dalla durata del ciclo di clock
- Il limite inferiore alla durata del clock è imposto dal passo di esecuzione più lento (tra i visti finora certamente quelli che prevedono il trasferimento attraverso l'ALU, come il ciclo T1 della fase di fetch)
- Ad ogni modo, le prestazioni dipendono anche dal numero medio C_{PI} di cicli richiesti per istruzione
- Per ridurre C_{PI} è necessario aumentare più possibile il parallelismo interno
- Ciò può essere ad esempio ottenuto aumentando il numero di percorsi indipendenti nella UO
- Consideriamo ad esempio la seguente architettura con tre bus interni alla UO:

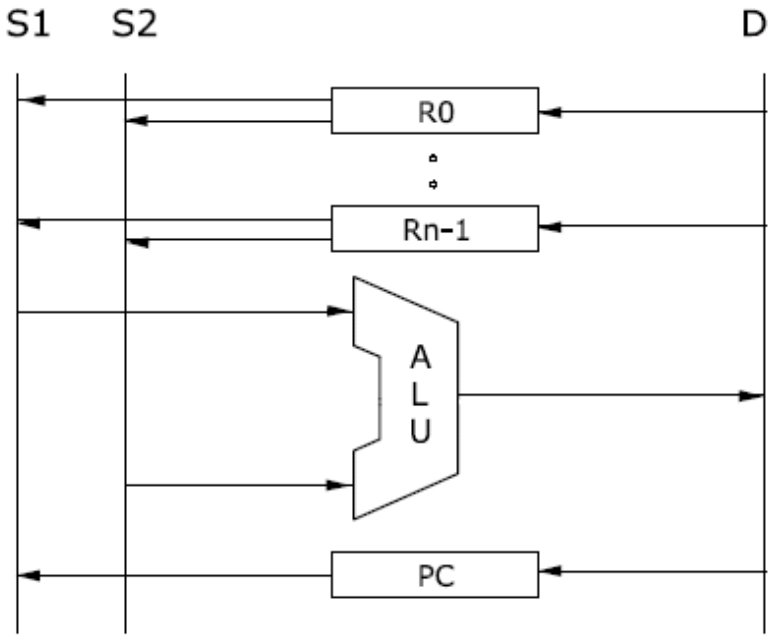
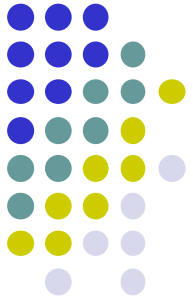


Figura 6.11 Architettura con tre bus interni. S1 e S2 fungono da *bus sorgenti*, D da *bus di destinazione*. L'istruzione `ADD RD,RA,RB` richiede un solo ciclo di clock per la fase di esecuzione.



- In tale architettura, l'istruzione *ADD RD, RA, RB,* invece di richiedere i 3 cicli di clock

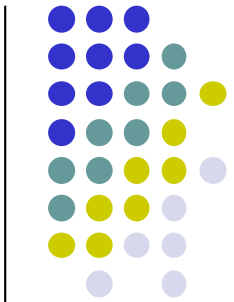
  T5: RA_{out} , TI_{in}
  T6: RB_{out} , *ADD,* TO_{in}
  T7: TO_{out} , RD_{in}

  richiederebbe soltanto il ciclo

  T6: RA_{out2S1}, RB_{out2S2}, ADD; RD_{in}

  dove RX_{out2Si} è il segnale di controllo che abilita l'uscita del registro *RX* sul bus *Si*

Realizzazione UC



- Ci sono due realizzazioni possibili della UC:
  - a logica cablata
  - a logica multiprogrammata
- Nella logica cablata, la UC è vista e progettata come una rete sequenziale sincrona
- Nella logica microprogrammata, la UC è vista come un calcolatore elementare all'interno del calcolatore in grado di eseguire *microistruzioni*
- Vediamo più in dettaglio le due realizzazioni

UC a logica cablata

- La UC è vista e progettata come una rete sequenziale sincrona in cui
  - ingressi: registro IR, stato della UO
  - uscite: segnali di controllo o comandi a UO, memoria e resto del sistema

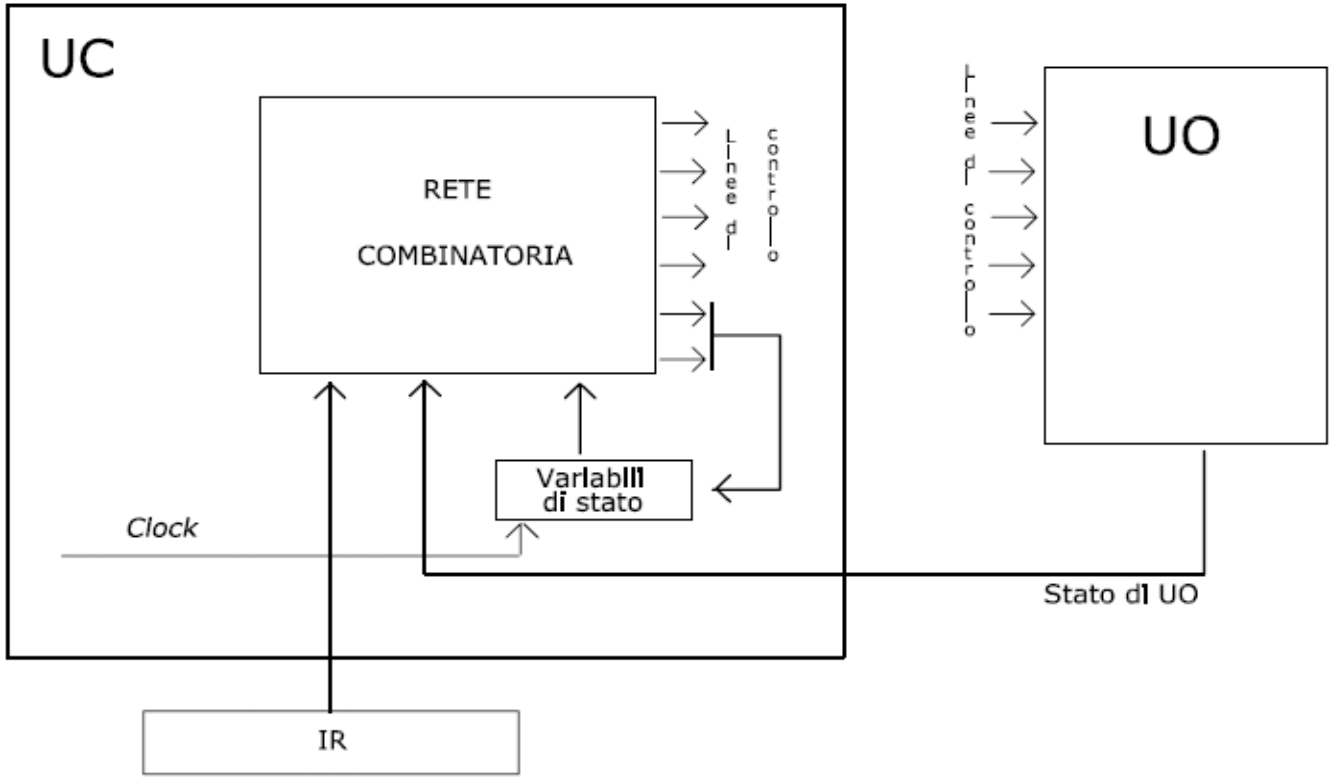
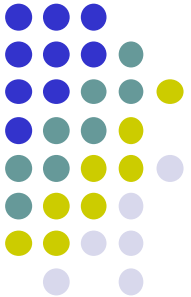


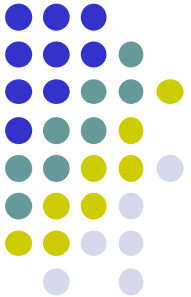
Figura 6.12 Unità di controllo a logica cablata.



- La sintesi è molto complessa a causa della grande complessità

- Misura della complessità di UC:
  N_stati x N_ingressi x N_uscite

- Viene realizzata mediante
  - ROM: la rete combinatoria ha come
    - ingressi (indirizzi alla ROM): IR, stato di UO, stato di UC
    - uscite: comandi, ingressi di eccitazione dei FF di stato
  - Logica programmabile (PLA)
  - Progettazione con CAD per VLSI



- Un approccio più modulare, invece di far ripartire il progetto della UC da capo, sfrutta le componenti sequenziali già sviluppate: registri, contatori, bus …

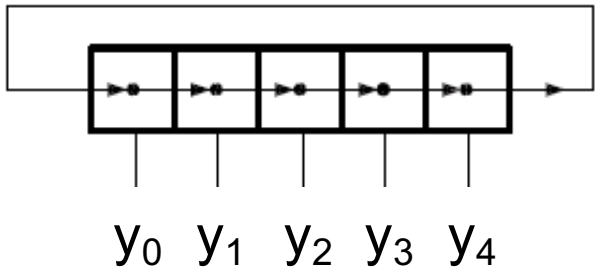
- In particolare, possiamo sfruttare il contatore ad anello che abbiamo già visto per generare i segnali di *temporizzazioneT1, T2* ,…, che identificano i successivi cicli di clock

- In pratica, *T1* è attivo durante il primo ciclo di clock di tutte le istruzioni, *T2* durante il secondo e così via

- A partire da tali segnali, è immediato ricavare le espressioni algebriche dei segnali di comando e selezione

Contatore 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



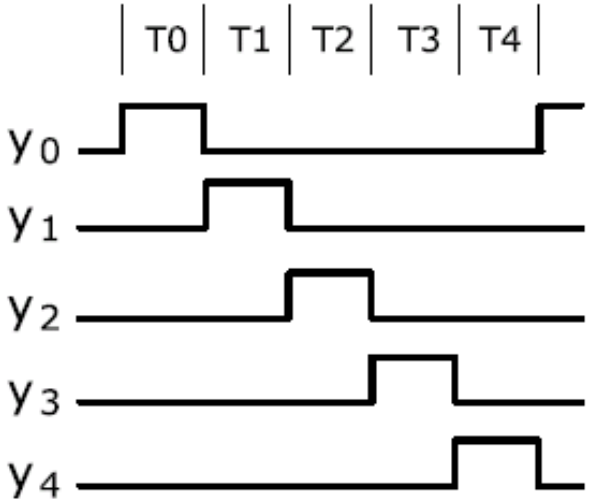
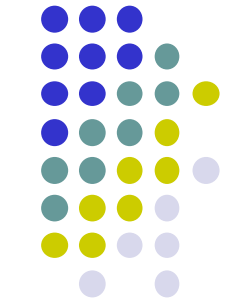


Figura 4.32 Registro ad anello (di 5 bit) e forme d'onda corrispondenti allo stato dei singoli bit. Il registro è precaricato in modo da contenere un solo 1.



- Ad esempio, nelle sequenze esaminate TO_{out} è attivo durante T2 (fase di fetch uguale per tutte le istruzioni) e durante T7 sia nella *ADD* che nella *ST*

- Quindi,

$$TO_{out}=T2+T7(ADD+ST)$$

  supponendo che i segnali *ADD* e *ST* vengano asseriti dalla UC quando nella fase di fetch vengono decodificate le istruzioni corrispondenti

- Vediamo la rete corrispondente:

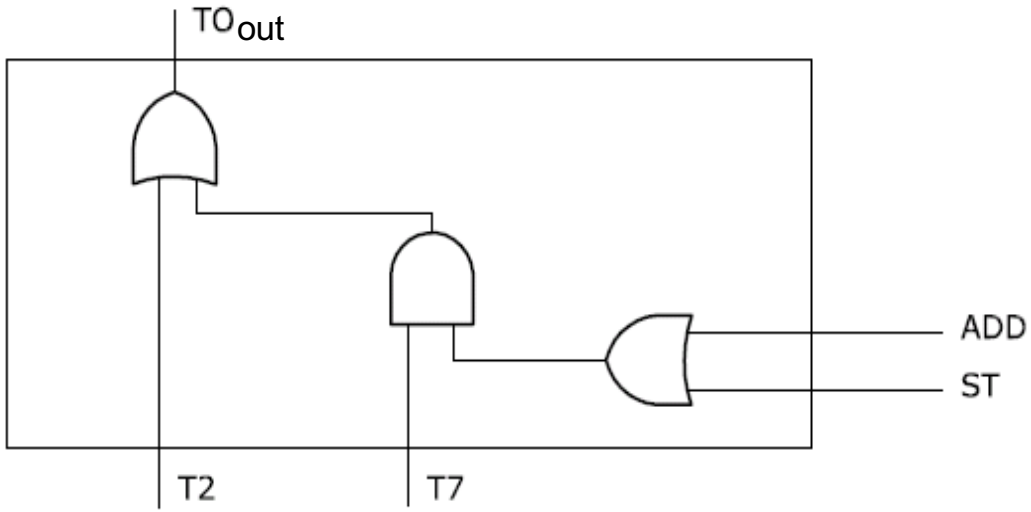
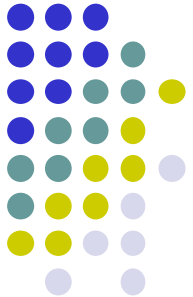


Figura 6.13 Rete per la generazione di TO.

- Similmente, sempre limitatamente alle sequenze già viste:

$$DTR_{out}=T4+(T9+T10)ST$$



- Il contatore ad anello deve generare un numero di segnali di temporizzazione *n* almeno pari alla lunghezza della fase di fetch più quella della più lunga fase di execute

- Poiché istruzioni diverse hanno fasi di execute diverse che possono durare un diverso numero di cicli, arricchiamo il nostro contatore ad anello con un ulteriore ingresso *Restart*, che quando asserito fa sì che al prossimo ciclo di clock si riparte da *T1*

- In riferimento alle sequenze viste

  $Restart = \ldots + T7 \cdot ADD + T10 \cdot ST + \ldots$

- Abbiamo quindi la rete seguente

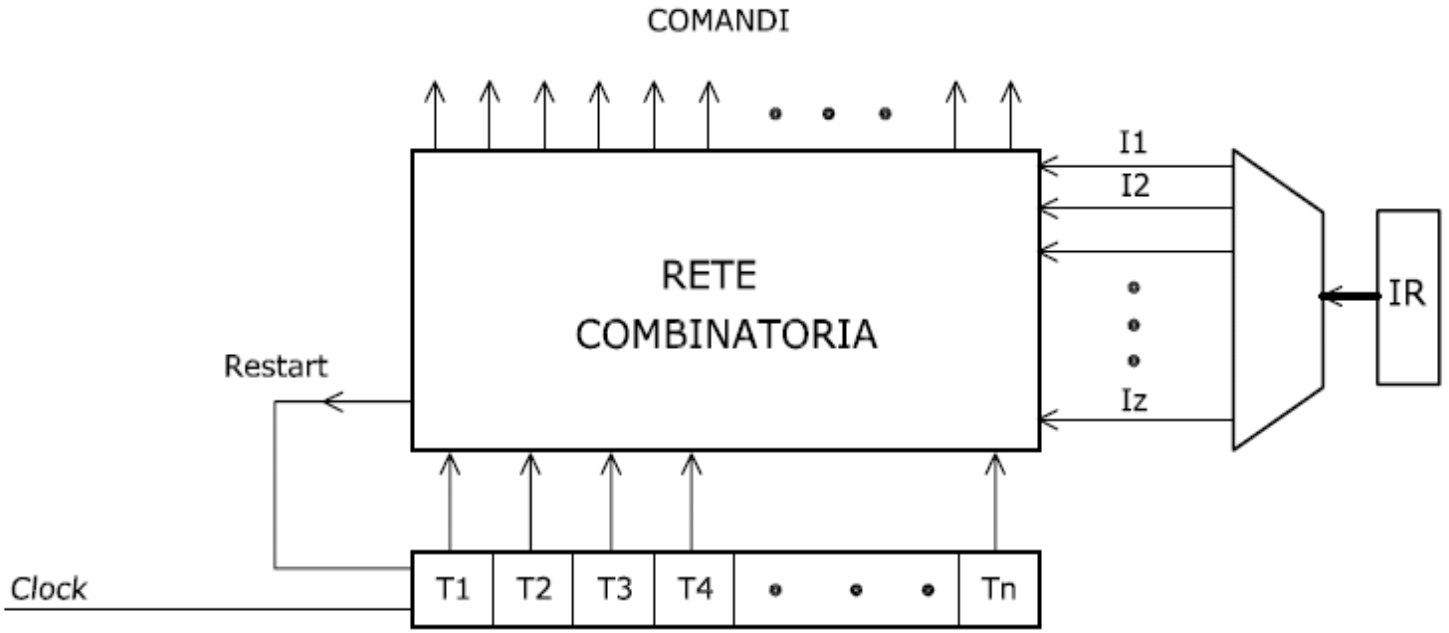


Figura 6.14 Rete per la generazione di comandi. Il decodificatore presenta z uscite, essendo z il numero di differenti codici di istruzione.

- Essa combina
  - i segnali di temporizzazione T1,…,Tn
  - i segnali di decodifica delle istruzioni I1,…,Iz
- In definitiva la UC che relega al contatore ad anello la parte sequenziale, mentre la logica è realizzata tramite una rete combinatoria