Istituto Tecnico Industriale Statale
"EUGENIO BARSANTI"

80038 POMIGLIANO D'ARCO (NA) Via Mauro Leone, 105
Tel. (081) 8841350 - Fax (081) 8841676
Distretto scolastico n. 31 - Cod. Fisc. 80104010634
Cod.Ist. NATF040003 - Cod. Serale NATF04050C
E-mail : NATF040003@istruzione.it

Specializzazioni: Meccanica
Elettrotecnica e Automazione
Elettronica e Telecomunicazioni
Informatica –Progetto Abacus
Corso Serale: Elettrotecnica e Automazione

DIPARTIMENTO DI ELETTRONICA

Proff. Mariano Riccardo - Paolo Bisconti - Paolo Rea

SISTEMI A MICROPROCESSORE

PARTE I: FONDAMENTI DELLA LOGICA PROGRAMMATA

APPUNTI DALLE LEZIONI
DEL CORSO DI SISTEMI

VERSIONE 09/2007

1. GENERALITA’ SUI SISTEMI A μP

1.1 Note introduttive

Il microprocessore (micro = piccolissimo; processore = elaboratore dati) è un dispositivo che, rispetto a tutti i circuiti logici combinatori e sequenziali studiati in Elettronica Digitale, ***non svolge una funzione logica prestabilita e descritta sinteticamente dalla relativa tabella di verità***.
Come se ne deduce dal nome, è un piccolissimo elaboratore di dati ovvero un Circuito Integrato (C.I.) il cui funzionamento è determinato da un “***programma***”. Il programma, (***il software***) è costituito da una successioni di ***istruzioni*** che vengono scritte in uno specifico linguaggio ed eseguite dal μP con la sequenza prestabilita.
Utilizzando C.I. combinatori e sequenziali si implementano circuiti “dedicati”, ovvero circuiti che svolgono solo la funzione per la quale sono stati progettati; utilizzando il microprocessore la situazione cambia radicalmente: lo stesso circuito (***hardware)*** può svolgere più funzioni diverse tra loro semplicemente modificando il programma (***software***).
Con l’impiego del microprocessore si passa dalla ***logica cablata*** alla ***logica programmata***.
E’ opportuno subito chiarire che il microprocessore non può operare da solo; per il ruolo che svolge può essere visto come un circuito che coordina il funzionamento di una serie di blocchi funzionali; per tale motivo viene comunemente chiamato “***Unità Centrale di Elaborazione*** ” ed indicato con l’abbreviazione ***CPU*** (Central Processing Unit).
L’insieme dei blocchi funzionali coordinati dalla CPU realizza fondamentalmente una struttura denominata ***sistema a microprocessore***. Tale struttura prende anche il nome di microcomputer o microcalcolatore; i Personal Computer sono dei sistemi a microprocessore.
Da un punto di vista logico un sistema a microprocessore risulta costituito dai seguenti blocchi funzionali:

- **CPU**
- **blocco di memoria (RAM e ROM)**
- **circuito di clock (CK)**
- **blocco di ingresso (IN)**
- **blocco di uscita (OUT)**

La fig. 1.1 rappresenta, nel modo più elementare, tale sistema mettendo in evidenza i ***versi*** dei ***flussi di dati***.

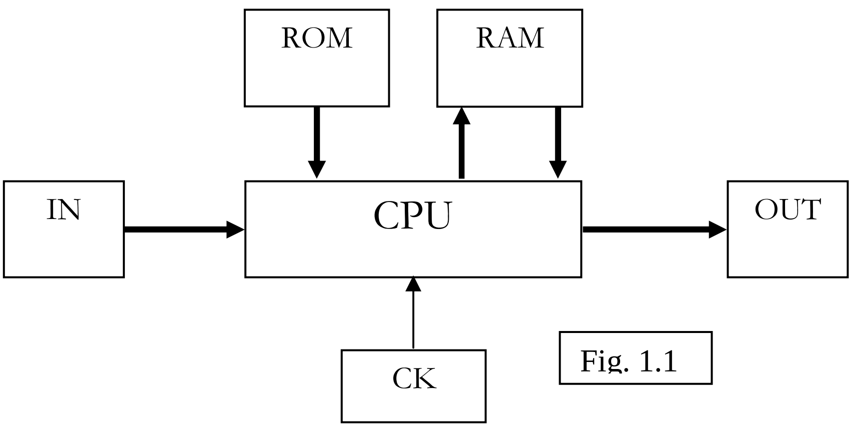


Fig. 1.1

Questi versi orientati ci consentono immediatamente di visualizzare le possibili tipologie di “colloquio” che possono avvenire nell’ambito del sistema.

Innanzitutto la CPU **legge** le istruzioni permanentemente memorizzate nella memoria ROM; inoltre essa, opportunamente "istruita" dal progettista del software, ***può eseguire***:

- la **lettura** di un dato **dalla memoria RAM**;
- la **scrittura** di un dato **nella memoria RAM**;
- la **lettura** di un dato **dal blocco di ingresso**;
- la **scrittura** di un dato **sul blocco di uscita.**

1.2 Il blocco di memoria

Come si nota dalla Fig. 1.1 il blocco di memoria si suddivide in due sottoblocchi: memoria ROM (**memoria di programma**) e memoria RAM (**memoria di dati**).
In linea di principio entrambe le memorie possono essere considerate come un insieme di "contenitori" in ognuno dei quali viene registrata, in forma binaria, una informazione; questi contenitori vengono chiamati ***locazioni di memoria***.
Nella memoria ROM è scritto, in modo permanente, il programma che la CPU deve eseguire; durante il funzionamento essa interagisce con la ROM svolgendo le seguenti due operazioni :

- "punta" in modo sequenziale tutte le locazioni di memoria ROM a partire dalla prima
- "preleva", cioè legge dalle locazioni ROM, sempre in modo sequenziale, le istruzioni una alla volta, le trasferisce al suo interno e le esegue.

Tutte le istruzioni che inizialmente sono state memorizzate nella ROM possono essere solo lette, per cui ***il flusso di dati è unidirezionale con direzione ROM → µP***
Viceversa nella memoria RAM la CPU può scrivere e quindi salvare, in modo temporaneo, dei dati, come per esempio quelli provenienti dal blocco di ingresso o quelli necessari per creare delle tabelle di codici; successivamente la CPU può leggere i dati salvati ed inviarli verso il blocco di uscita.
In questo caso ***il flusso di dati è bidirezionale.***

1.3 Il circuito di clock

Il µP legge in sequenza le singole istruzioni che sono memorizzate nella memoria ROM e le esegue con una certa velocità.
Il circuito di clock è un generatore di segnale ad onda quadra la cui frequenza determina la "velocità" con la quale il microprocessore esegue le singole istruzioni costituenti il programma. Tale circuito, inoltre, sincronizza tutte le operazioni interne al sistema legate all'esecuzione delle istruzioni.

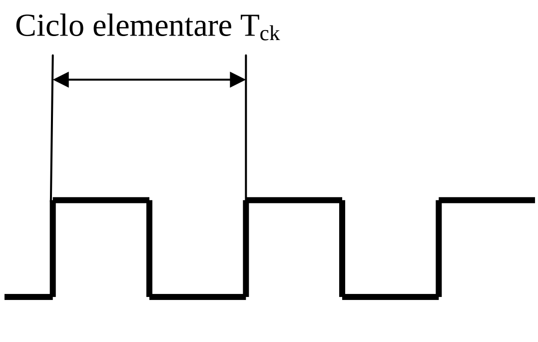


Fig. 1.2

Il segnale di clock (CK) generato dall'omonimo circuito è rappresentato in fig. 1.2 ed è caratterizzato dal ***periodo elementare*** o ***ciclo elementare T_{ck}***.
Il sistema sperimentale che utilizziamo nel nostro laboratorio è costituito da un µP che lavora con una frequenza ***f = 2,5 MHz*** per cui il ***periodo elementare*** o ***ciclo elementare T_{ck}*** risulta essere: ***T_{ck} = 1/f = 0,4µs***.
A puro titolo di confronto, le frequenze di lavoro dei µP che attualmente sono a corredo dei Personal

Computer, raggiungono valori il cui ordine di grandezza è pari a migliaia di MHz (GHz).
Perché allora continuiamo a studiare lo Z80? Le ragioni sono molteplici: innanzitutto perché è molto semplice da capire (quasi banale rispetto ai microprocessori dell'ultima generazione); poi perché è più che sufficiente, in termini di capacità e velocità di calcolo, per il nostro obiettivo principale, il controllo automatico dei sistemi (salvo casi eccezionali, infatti, le grandezze fisiche da controllare variano con una velocità tale da richiedere tempi di intervento dell'ordine dei ms); infine perché la conoscenza dello Z80 può essere propedeutica per lo studio dei microcontrollori, che tanta diffusione attualmente stanno avendo.

1.4 Le istruzioni

Ogni microprocessore presenta un certo numero di istruzioni che può eseguire ed il loro insieme prende il nome di "set delle istruzioni"
Ciascuna delle singole istruzioni è formata da:

- un ***codice operativo*** che rappresenta "***cosa deve fare la CPU***", ovvero l'operazione che essa deve eseguire
- un ***operando*** che può essere un ***dato*** sul quale si esegue l'operazione oppure un ***indirizzo*** di una locazione di memoria o di un periferico di I/O che costituiscono il sistema.

Ogni singola istruzione viene eseguita in due fasi:

- la prima fase, detta ***fase di fetch*** (andare a prendere), ***è la fase di lettura in ROM del codice operativo.***
- la seconda fase, detta ***fase di execute*** (eseguire), ***è la fase di esecuzione dell'istruzione letta.***

La fig. 1.3 mostra ***in modo qualitativo*** l'andamento temporale di un ciclo di istruzione. Di seguito si evidenziano gli scambi di informazioni che avvengono tra CPU e ROM durante la fase di fetch:

- il primo scambio CPU → ROM riguarda il ***puntamento*** della locazione di memoria nella quale è contenuto il codice operativo;
- il secondo scambio ROM → CPU riguarda il trasferimento di questo codice dalla locazione di memoria in un particolare registro interno della CPU.

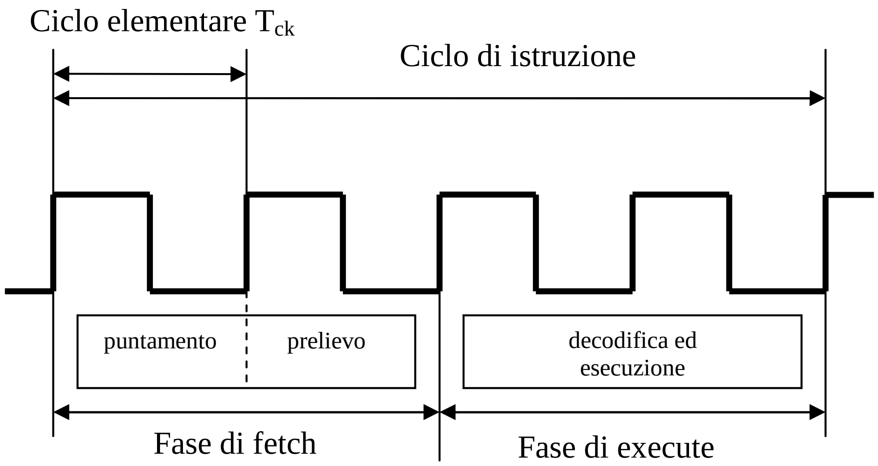


Fig. 1.3

Terminata la fase di fetch, inizia quella di execute: il codice operativo, che è stato trasferito nella CPU, viene opportunamente decodificato (decifrato) e l'istruzione viene eseguita. Eseguire l'istruzione può voler dire anche ritornare nella ROM per prelevare un operando (un indirizzo o un numero) su cui effettuare l'operazione prevista.
L'insieme di un certo numero di istruzioni costituisce il programma che la CPU deve svolgere.
Tutte le istruzioni eseguibili da un µP sono composte da ***uno o più cicli standard*** denominati ***cicli macchina*** che, per essere portati a termine, richiedono un tempo pari a ***3 o 4 cicli elementari*** $\boldsymbol{T_{ck}}$.
Questo significa che ogni ciclo di istruzione necessita di un certo numero di cicli elementari T_{ck} il

cui numero è riportato in apposite tabelle che raccolgono il set di istruzioni del µP in esame; è allora possibile calcolare i tempi richiesti dalla CPU per l'esecuzione di ogni istruzione. Riferendoci ancora alla fig. 1.3, l'istruzione assunta come esempio viene eseguita in un ciclo composto da 4 cicli elementari T_{ck} per cui viene eseguita in un tempo pari a 0,4 x 4 = 1,6 µs. (Ricordiamo che nel nostro sistema di sviluppo per Z80 $\boldsymbol{T_{ck} = 1/f = 0{,}4\mu s}$).
Alla luce di quando abbiamo precisato si può ora affermare che per programma permanentemente memorizzato in una memoria ROM, dobbiamo intendere che ***in ogni locazione di memoria sono stati salvati i codici operativi e gli eventuali operandi di ogni singola istruzione***.

1.5 I blocchi di ingresso ed uscita

Genericamente possiamo definire i blocchi di ingresso e di uscita nel seguente modo:

- Il blocco di ingresso rappresenta il canale di comunicazione attraverso il quale il µP riceve i dati provenienti dal "mondo esterno" per poterli elaborare secondo un determinato programma. ***Il flusso di dati è unidirezionale con direzione IN → µP***

- Il blocco di uscita rappresenta il canale di comunicazione attraverso il quale il mondo esterno riceve i dati elaborati dal µP. ***Il flusso di dati è unidirezionale con direzione µP → OUT***

Come mostra la fig. 1.4, un blocco di ingresso è sempre un dispositivo trasmettitore (TX), un blocco di uscita è sempre un dispositivo ricevitore (RX), mentre il µP può svolgere le funzioni di RX o TX a seconda che colloqui rispettivamente con periferici di ingresso o di uscita.

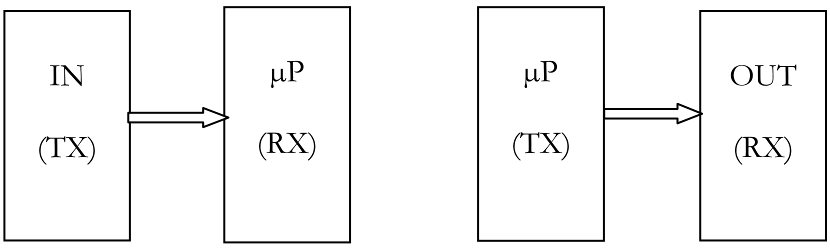


Fig. 1.4

Nello studio dei sistemi a µP è concettualmente utile scomporre sia il blocco di IN che quello di OUT in due sottoblocchi come mostrato in fig. 1.5.

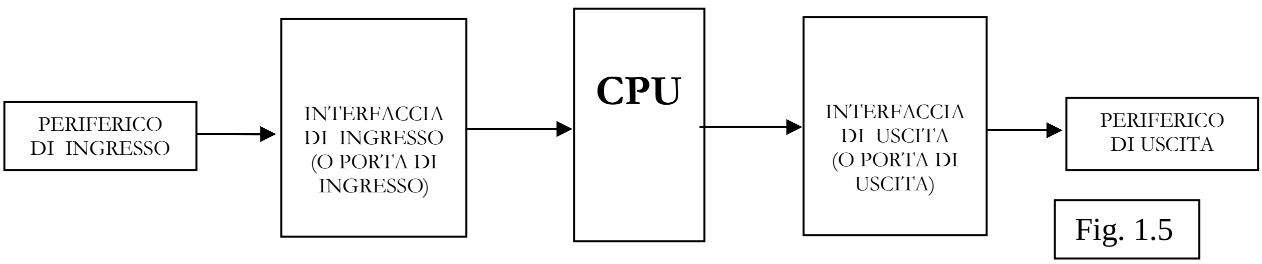


Fig. 1.5

L'analisi della fig. 1.5 ci consente di precisare che:

- ***il periferico di ingresso*** è inteso come quel dispositivo che materialmente rende disponibile alla sua uscita il dato binario da inviare alla CPU;
- ***l'interfaccia di ingresso (o porta di ingresso)*** è quel circuito che fisicamente realizza il collegamento tra l'uscita del periferico di ingresso e la CPU, consentendo il trasferimento del dato binario e la relativa sincronizzazione; di solito è costituita da un connettore e da un C.I. che abiliti o blocchi l'accesso del dato verso il µP.
- ***l'interfaccia di uscita (o porta di uscita)*** è quel circuito che fisicamente realizza il collegamento tra la CPU e l'ingresso del periferico di uscita, consentendo il trasferimento del dato binario e la relativa sincronizzazione; di solito è costituita da un connettore e da un C.I. che memorizza temporaneamente il dato per tenerlo a disposizione del periferico di uscita, consentendo al µP di dedicarsi ad altre operazioni.
- ***il periferico di uscita*** è inteso come quel dispositivo che materialmente utilizza il dato binario proveniente dalla CPU.

La CPU con le sue interfacce di ingresso e di uscita rappresenta, nell'ambito di un sistema a microprocessore, il ***sistema di elaborazione e controllo*** che gestisce i dati provenienti dal periferico di ingresso e li invia in uscita per pilotare il periferico di uscita.

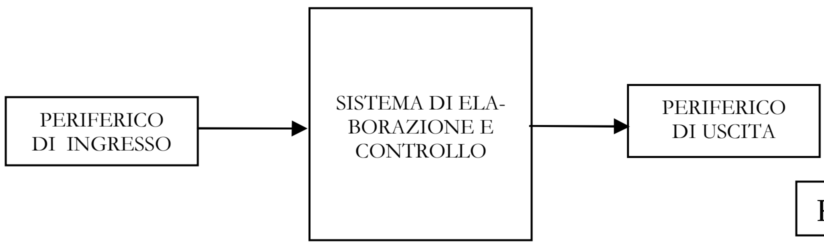


Fig. 1.6

I periferici di ingresso ed uscita (periferici di I/O) sono molto diversi tra loro sia per la funzione svolta che per le caratteristiche funzionali.
I periferici di I/O normalmente presenti in un sistema Personal Computer sono: tastiera, mouse, scanner, ecc. come dispositivi di ingresso, e video, stampante, plotter, ecc. come dispositivi di uscita.
I periferici di I/O normalmente presenti in laboratorio ed impiegati ***in fase di simulazione e collaudo*** dei sistemi a microprocessore sono: pulsanti, switch e diodi led.
Pulsanti e switch sono dispositivi di input che vengono utilizzati per simulare uscite digitali di periferici di ingresso più complessi; in particolare i pulsanti simulano segnali di tipo impulsivo mentre con n switch è possibile simulare un dato binario in ingresso ad n bit.
I diodi led sono dispositivi di output che vengono utilizzati per rendere visibile il risultato dell'elaborazione; in particolare con la loro accensione possono simulare la attivazione di alcune linee di comando oppure, utilizzando 8 diodi led, è possibile rendere visibile un dato binario di uscita ad 8 bit.
Per quelle che sono le finalità del Corso, particolare importanza assumono le problematiche legate al ***controllo di processo*** intendendo per esso un sistema a µP che svolge le tipiche funzioni di ***acquisizione dati***, loro elaborazione e ***distribuzione dati.***

In questo caso particolare, con riferimento alla fig. 1.6,

- il blocco “periferico di ingresso” è un insieme di sottoblocchi funzionali che nel complesso realizza l’acquisizione dati;
- il blocco “periferico di uscita” è un insieme di sottoblocchi funzionali che nel complesso realizza la distribuzione dati.

Si ottiene pertanto la rappresentazione di fig. 1.7

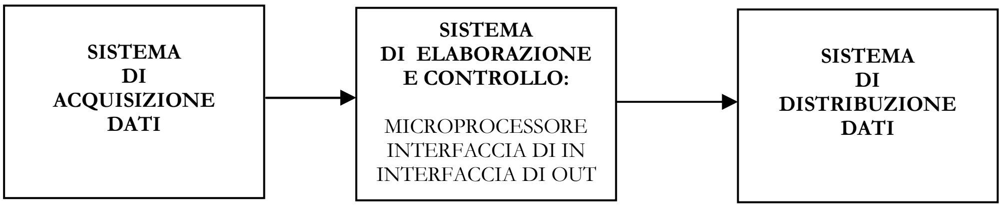


Fig. 1.7

2. MEMORIE

2.1 Generalità.

Le memorie sono circuiti elettronici in grado di "conservare" dei dati espressi in forma binaria. Il "contenitore" nel quale è memorizzato un dato viene chiamato ***locazione di memoria***. Normalmente, per le memorie a corredo dei piccoli sistemi a µP, i dati binari contenuti in ogni locazione sono costituiti da gruppi di otto bit = 1 byte.
Il ***numero di locazioni*** presenti in una memoria ci fa capire quanto è "grande" una memoria; questo numero prende il nome di ***capacità di memoria*** .
L'unità di misura della capacità di memoria è 1K corrispondente a 1024 locazioni di memoria.
Una memoria di nK contiene n*1024 locazioni di memoria.
L'operazione di ***memorizzazione*** di un dato in una locazione di memoria è chiamata ***scrittura*** mentre l'operazione di ***prelievo*** del dato memorizzato in una locazione di memoria è chiamata ***lettura***.
In merito alla memorizzazione dei dati può essere fatta una prima classificazione, distinguendo le memorie in:

- ***memorie volatili:***: sono memorie in cui il dato è presente in modo ***temporaneo*** nel senso che viene perso nel momento in cui viene a mancare l'alimentazione.
- ***memorie non volatili:***: sono memorie in cui il dato è presente in modo ***permanente*** nel senso che non viene perso nel momento in cui viene a mancare l'alimentazione.

In merito alla scrittura e lettura dei dati può essere fatta una seconda classificazione, distinguendo le memorie in :

- ***memorie a sola lettura:*** sono memorie in cui i dati originari contenuti, possono solo essere letti
- ***memorie a lettura e scrittura:*** sono memorie in cui i dati possono essere letti e scritti.

Una ulteriore classificazione delle memorie può essere fatta in relazione alla loro ***struttura interna***; in base ad essa si distinguono:

- ***memorie ad accesso sequenziale:*** l'accesso ad un determinato dato è di tipo sequenziale, nel senso che per arrivare al dato è necessario fare scorrere tutti i dati che lo precedono. Sono poco usate e non verranno considerate in questa trattazione.
- ***memorie ad accesso casuale:*** in questo caso l'accesso al dato è diretto.

2.2 Memorie ad accesso casuale.

Nei sistemi a microprocessore le memorie che si utilizzano hanno una struttura ad accesso casuale, cioè consentono, come già detto, l'accesso diretto alla locazione interessata.
Tale struttura può essere rappresentata con una matrice di locazioni di memoria, costituita da una serie di righe e colonne, come mostrato dalla fig. 2.1.

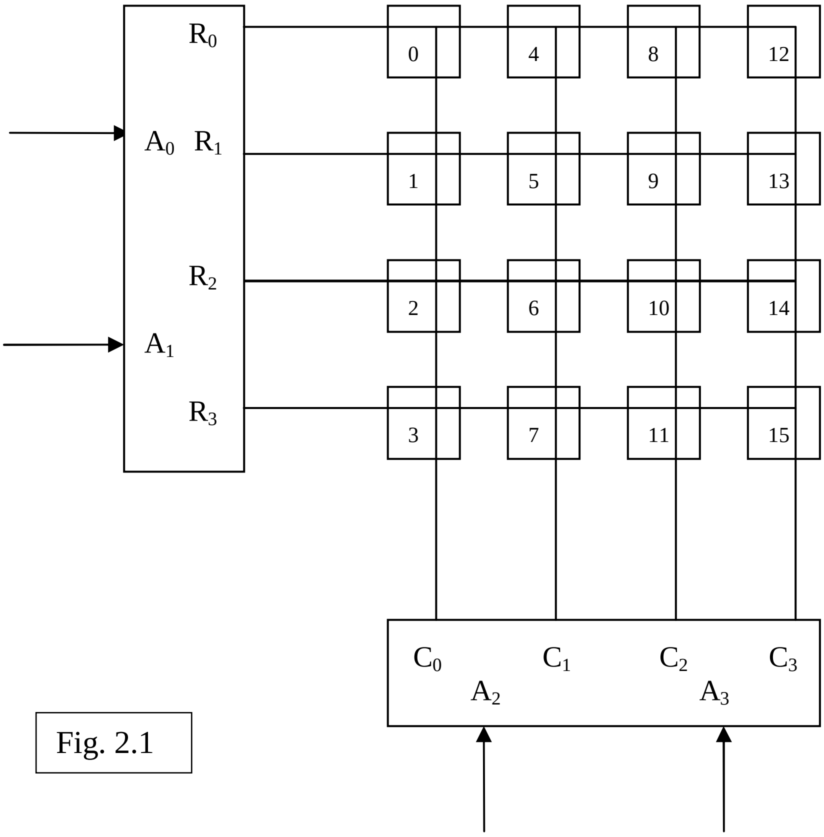


Fig. 2.1

Per tale tipo di struttura la gestione di scrittura e lettura dei dati risulta completamente diversa da quella delle memorie ad accesso sequenziale.

Nell'esempio di fig. 2.1 sono rappresentate 16 locazioni di memoria disposte secondo le righe R_0 R_1 R_2 R_3 e secondo le colonne C_0 C_1 C_2 C_3. Le righe possono essere selezionate in base agli ingressi A_0 ed A_1 del decodificatore di riga ed analogamente le colonne possono essere selezionate in base agli ingressi A_2 ed A_3 del decodificatore di colonna.
Le tabelle di verità dei due decodificatori sono le seguenti.

A_1	A_0	U
0	0	R_0
0	1	R_1
1	0	R_2
1	1	R_3

A_3	A_2	U
0	0	C_0
0	1	C_1
1	0	C_2
1	1	C_3

In questo modo ogni singola locazione di memoria può essere individuata dalle sue "coordinate" (riga e colonna) ed ***il tempo di lettura risulta sempre lo stesso***, qualunque sia la locazione indirizzata.

Come esempio, supponiamo di voler indirizzare la locazione n° 9 di fig. 2.1; essa è individuata dalla riga R_1 e dalla colonna C_2 . In base alle tabelle di verità, la riga R_1 e la colonna C_2 sono selezionate dai valori: $A_1 = 0$; $A_0 = 1$; $A_3 = 1$; $A_2 = 0$ e pertanto ***l'indirizzo*** della locazione di memoria n° 9 è : **A_3 A_2 A_1 A_0 = 1 0 0 1**. Si noti che l'indirizzo coincide con il numero assegnato alla locazione.

Per questo tipo di memorie ogni locazione è identificata attraverso il suo indirizzo.

Continuando a riferirsi all'esempio di Fig. 2.1, facciamo la seguente considerazione: per selezionare una memoria di 16 locazioni, è necessario disporre di 4 linee di indirizzo. E' facile intuire che se si avessero decodificatori di riga e di colonna con tre ingressi (ovvero 6 linee di indirizzo), essi avrebbero otto uscite ed in tal modo si potrebbero indirizzare 8 x 8 = 64 locazioni di memoria. Se i decodificatori avessero quattro ingressi (ovvero 8 linee di indirizzo), presenterebbero 16 uscite e potrebbero indirizzare 16 x 16 = 256 locazioni di memoria

In definitiva ***esiste una corrispondenza tra numero di linee di indirizzo e la capacità di memoria :***

- 4 linee di indirizzo = 16 locazioni
- 5 linee di indirizzo = 32 locazioni
- 6 linee di indirizzo = 64 locazioni
- 7 linee di indirizzo = 128 locazioni
- 8 linee di indirizzo = 256 locazioni
- 9 linee di indirizzo = 512 locazioni
- **10 linee di indirizzo = 1024 locazioni = 1K (unità di misura della capacità di memoria)**
- 11 linee di indirizzo = 2048 locazioni = 2K
- 12 linee di indirizzo = 4096 locazioni = 4K
- 13 linee di indirizzo = 8192 locazioni = 8K e così via.

In generale, con n linee di indirizzo si possono indirizzare 2^n locazioni di memoria.

Ogni locazione di memoria riceve, in fase di scrittura, o invia, in fase di lettura, i dati, ovvero gli 8 bit, attraverso 8 distinte linee che sono le linee dati della singola locazione; pertanto, nel suo complesso, la matrice di locazioni di memoria può essere vista come un unico blocco funzionale che presenta:

- "n" linee di indirizzo in relazione alla propria capacità di memoria
- 8 linee dati, nell'ipotesi che i dati contenuti nelle singole locazioni siano ad 8 bit.

Con queste premesse è possibile rappresentare con lo schema a blocchi di fig. 2.2 un C.I. di memoria ad accesso casuale.

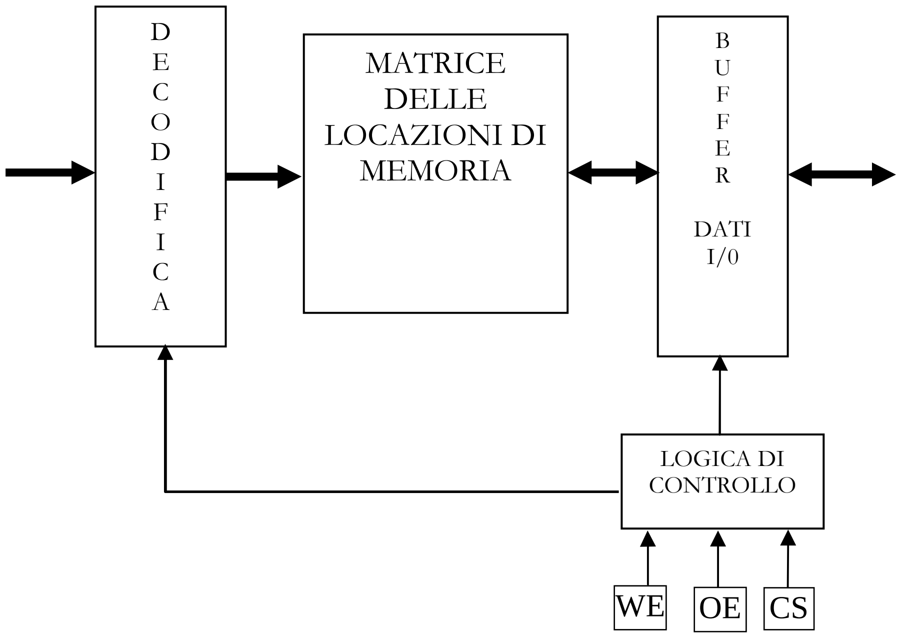


Fig. 2.2

La linea CS della logica di controllo consente di attivare o disattivare l'intero dispositivo.

Le linee WE e OE, sempre della logica di controllo, consentono di selezionare le due distinte operazioni di scrittura e lettura dei dati.

2.3 Memorie ROM

Le memorie ROM (Read Only Memory ovvero memoria a sola lettura) sono ***memorie ad accesso casuale, non volatili, a sola lettura***.
Essendo memorie a sola lettura consentono unicamente il prelievo dei dati memorizzati; inoltre, essendo memorie non volatili, tutti i dati in essa contenuti non vengono persi all'atto dello "spegnimento" del sistema.
E' necessario fare una classificazione tra le diverse memorie di tipo ROM; in modo sintetico queste memorie si possono dividere in:

- ***ROM mask*** (a maschera)
- ***ROM programmabili***. (***EPROM - EEPROM - FLASH)***

Le memorie ROM mask, comunemente chiamate semplicemente ROM, sono memorie il cui contenuto viene scritto all'origine dal costruttore e non può più essere modificato. Il loro campo di impiego è di tipo specialistico, nel senso che sono programmate ed immesse sul mercato per svolgere un'unica funzione; così in commercio si trovano ROM convertitrici di codici, ROM generatori di caratteri, ROM contenenti tabelle di verità relative a circuiti logici con elevato numero di variabili ecc. Nei primi Personal Computer questa memoria veniva utilizzata per la memorizzazione del BIOS (Basic Input Output Services).
Le memorie EPROM differiscono dalle ROM mask per il loro funzionamento più flessibile nel senso che possono essere programmate e cancellate più volte.
Una EPROM viene fornita dal costruttore ***non programmata*** e può essere scritta utilizzando un apposito dispositivo chiamato "***programmatore di EPROM***".
Ove ce ne fosse bisogno, è possibile cancellare il contenuto con l'ausilio di un altro dispositivo chiamato "***cancellatore di EPROM*** "; esso è dotato di una particolare lampada a raggi ultravioletti che vengono inviati all'integrato attraverso una apposita finestra trasparente posta sul contenitore. E' importante precisare che se si volesse modificare anche un solo bit memorizzato, bisognerebbe procedere ad una cancellazione completa
Le EPROM presentano i seguenti inconvenienti:

- rimozione del chip dal circuito ogni qualvolta è necessario procedere all'aggiornamento del contenuto;
- la necessità di impiego del programmatore e del cancellatore di EPROM
- tempi di cancellazione molto lunghi (20 – 25 minuti)

Nonostante tali inconvenienti le EPROM trovano ancora oggi largo impiego nei sistemi didattici e nella realizzazione dei prototipi per il costo contenuto.
Le memorie EEPROM (o E^2PROM) differiscono dalle EPROM in quanto cancellabili elettricamente: ciò consente di ottenere tempi di cancellazione notevolmente più piccoli; inoltre la cancellazione e successiva programmazione possono essere di tipo selettivo, nel senso che si può operare anche solo su una parte dei dati in memoria.
Infine le memorie denominate FLASH sono simili alle EEPROM avendo, rispetto ad esse, oltre che tempi di cancellazione e scrittura più rapidi, la possibilità di aggiornare il loro contenuto direttamente sul sistema in cui sono inserite.

La ***rappresentazione funzionale*** di una ROM è quella riportata in fig. 2.3. (D'ora in avanti considereremo solo memorie ad 8 bit, cioè capaci di memorizzare dati costituiti da un byte)
Sono presenti le seguenti linee:

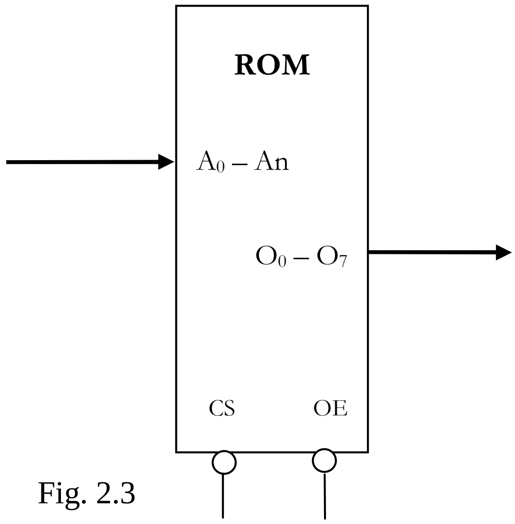


Fig. 2.3

- ***Linee A_0 - A_n di indirizzo;*** per quanto detto in precedenza il numero di tali linee dipende dalla capacità di memoria. Una ROM da 1K avrà 10 linee di indirizzo $A_0 – A_9$; una ROM da ¼ K avrà 8 linee di indirizzo $A_0 – A_7$; una ROM da 4K avrà 12 linee di indirizzo $A_0 – A_{11}$ e così via.
- ***Linee $O_0 – O_7$ dei dati;*** se ipotizziamo che in ogni locazioni di memoria il dato contenuto sia un byte le linee dati saranno $O_0 – O_7$.
- ***Linea CS di abilitazione del circuito;*** questa linea (selezione del chip) consente di abilitare il funzionamento del circuito integrato. In particolare quando su tale linea è presente un valore logico alto, il C.I. non è abilitato al funzionamento mentre quando è presente un valore logico basso il C.I. risulta abilitato al funzionamento (***CS è attiva bassa***).
- ***Linea OE di abilitazione alla lettura;*** un livello logico basso presente su tale linea determina l'abilitazione all'uscita dei dati e quindi consente l'operazione di lettura. Un livello logico alto presente su tale linea determina, anche con C.I. attivo, uno stato di alta impedenza per le linee $O_0 – O_7$.

Nota la piedinatura di una ROM è possibile descrivere come avviene la ***lettura di un dato.***

a) Si imposta l'indirizzo
b) Si abilita il C.I. mettendo a massa CS
c) Si legge il dato attivando OE

Risulta importantissima la seguente osservazione: a parte il breve intervallo di tempo durante il quale sulle linee O_0 - O_7 è presente il dato da leggere, ***tali linee si trovano in uno stato di alta impedenza***; questa condizione permetterà di poter collegare le suddette linee sul bus dati del µP senza la necessità di prevedere alcun tipo di interfaccia.

2.4 Memorie RAM

Le memorie RAM (Random Access Memory ovvero memorie ad accesso casuale) sono ***memorie ad accesso casuale, volatili, a scrittura e lettura***. Essendo memorie a scrittura e lettura consentono il caricamento ed il prelievo dei dati necessari per lo svolgimento del programma; inoltre, essendo memorie volatili, tutti i dati vengono persi all'atto dello "spegnimento" del sistema.
Senza voler entrare nel merito di come vengono tecnologicamente realizzate le singole celle di memoria, concettualmente esistono due tipologie di RAM: ***statiche*** e ***dinamiche***.
La cella elementare di memoria di una RAM statica (Static RAM = SRAM) può essere considerata come un Flip Flop tipo D: una volta memorizzato il bit 0 o 1 tale dato non viene perso, sempre che l'alimentazione non venga a mancare.
La cella elementare di memoria di una RAM dinamica (DRAM) può invece essere considerata come un condensatore: in questo caso uno 0 corrisponde al condensatore scarico mentre un 1 al condensatore carico. Il condensatore carico tende a scaricarsi e ciò determina la perdita di informazione; per evitare questo problema bisogna provvedere, periodicamente, a ripristinare la carica. Questo intervento periodico di ricarica prende il nome di ***rinfresco***. La denominazione "dinamica" nasce proprio dal fatto che, per tali memorie, deve essere continuamente eseguita l'operazione di rinfresco per ogni singola cella.
La necessità di una continua operazione di rinfresco rende le memorie dinamiche circuitalmente più complesse rispetto a quelle statiche; il vantaggio sta nel fatto che le DRAM offrono una possibilità di integrazione maggiore rispetto alle SRAM e ciò consente la realizzazione di elevate capacità di memoria con costi inferiori a quelli che si avrebbero, a parità di capacità, usando una tecnologia statica.
Normalmente, per sistemi che non richiedono grandi capacità di memoria, vengono sicuramente impiegare RAM statiche mentre nei sistemi che necessitano di elevate capacità di memoria si ricorre alle RAM dinamiche.
La ***rappresentazione funzionale*** di una RAM è riportata in fig. 2.4

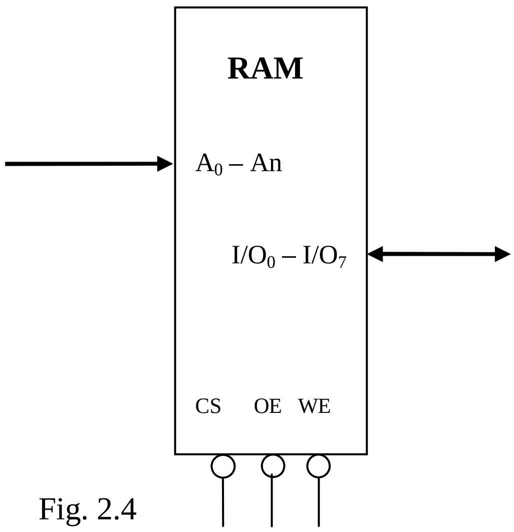


Fig. 2.4

Sono presenti le seguenti linee :

- ***Linee A_0 - A_n di indirizzo***; per quanto detto in precedenza il numero di tali linee dipende dalla capacità di memoria della RAM. Una RAM da 1K avrà 10 linee di indirizzo $A_0 – A_9$; una RAM da ¼ K avrà 8 linee di indirizzo $A_0 – A_7$; una RAM da 4K avrà 12 linee di indirizzo $A_0 – A_{11}$ e così via.
- ***Linee $I/O_0 – I/O_7$ dei dati***; se ipotizziamo che in ogni locazioni di memoria il dato contenuto sia un byte, le linee dati saranno otto: $I/O_0 – I/O_7$
- ***Linea CS di abilitazione del circuito***; questa linea (selezione del chip) consente di abilitare il funzionamento del circuito integrato. In particolare quando su tale linea è presente un

valore logico alto, il C.I. non è abilitato al funzionamento mentre quando è presente un valore logico basso il C.I. risulta abilitato al funzionamento (**C*S è attiva bassa***).

- ***Linea OE di abilitazione alla lettura***; un livello logico basso presente su tale linea determina l'abilitazione all'uscita dei dati e quindi consente l'operazione di lettura. Un livello logico alto presente su tale linea determina, con C.I. attivo, uno stato di alta impedenza per le linee $I/O_0 – I/O_7$
- ***Linea WE di abilitazione alla scrittura***; un livello logico basso presente su tale linea determina l'abilitazione dell'operazione di scrittura.

Rispetto ad una EPROM nelle RAM troviamo anche l'ingresso WE di abilitazione alla scrittura.
Nota la piedinatura di una RAM è possibile descrivere come avvengono le operazioni di ***lettura e scrittura di un dato.***

<u>Lettura</u>

a) Si imposta l'indirizzo
b) Si abilita il C.I. mettendo a massa CS
c) Si legge il dato Attivando OE

<u>Scrittura</u>

a) Si imposta l'indirizzo
b) Si imposta il dato
c) Si abilita il C.I. mettendo a massa CS
d) Si scrive il dato attivando WE

Anche per la RAM, come per la ROM, risulta importantissima la seguente osservazione: a parte il breve intervallo di tempo durante il quale sulle linee O_0 - O_7 è presente il dato da leggere, ***tali linee si trovano in uno stato di alta impedenza***; questa condizione permetterà, di poter collegare le suddette linee sul bus dati del µP senza la necessità di prevedere alcun tipo di interfaccia.

2.5 Memorie EPROM e RAM commerciali

I Circuiti Integrati relativi alle memorie commerciali vengono identificati dai costruttori come riportato nei seguenti esempi:

- 16 K (2K x 8 bit)
- 64 K (8K x 8 bit)

Si nota come commercialmente venga dichiarata la capacità complessiva in termini di bit. Viene poi specificato tra parentesi la capacità di memoria intesa come numero di locazioni e la relativa lunghezza di parola che è contenuta in ogni singola locazione.

Nel nostro laboratorio come EPROM sono disponibili, tra l'altro, i **C.I. 27C32 e 27C64**.
Il C.I. **27C64**, riportato in fig. 2.5, è una EPROM da 8Kbyte a 28 piedini.
Sono presenti le seguenti linee:

- 13 linee di indirizzo $A_0 – A_{12}$
- 8 linee dati $O_0 – O_7$
- una linea di alimentazione Vcc
- una linea di alimentazione Vpp
- una linea di massa Vss
- una linea di abilitazione CE (Chip Enable), attiva bassa
- una linea OE (Output Enable), attiva bassa
- una linea PGM (Program Enable) attiva bassa
- una linea NC (No Connection) non collegata internamente la cui presenza serve solo per rendere pari il numero di pin.

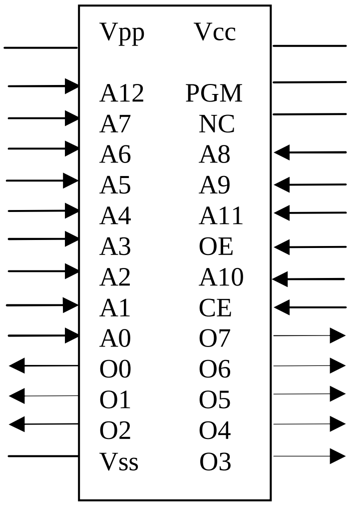


Fig. 2.5 - CMOS EPROM 8K * 8
27C64

Sulle linee Vpp e PMG, in fase di programmazione, vengono automaticamente applicati i valori di tensione necessari; in fase di lavoro questi due terminali vengono posti a +5V.

Il C.I. **27C32** è una EPROM da 4Kbyte a 24 piedini; sono presenti le seguenti linee:

- 12 linee di indirizzo $A_0 – A_{11}$
- 8 linee dati $O_0 – O_7$
- una linea di alimentazione Vcc
- una linea di massa Vss
- una linea CS di abilitazione (attiva bassa)
- una linea OE/Vpp (OE attivo basso)

Rispetto al 27C64 mancano le linee A_{12} , Vpp, PGM e NC
La linea OE/Vpp, in fase di programmazione, riceve automaticamente dal programmatore di EPROM la tensione necessaria per la memorizzazione dei dati. Durante l'uso, ovvero in lettura, è una linea di abilitazione.

Come memoria RAM è disponibile l'integrato 6116 riportato nella fig. 2.6;

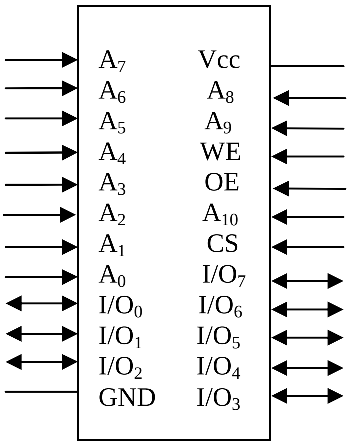


Fig. 2.6 - CMOS STATIC RAM 2K * 8
6116

Si tratta di una RAM statica da 2 Kbyte a 24 piedini.

Sono presenti le seguenti linee:

- 11 linee indirizzo $A_0 – A_{10}$
- 8 linee dati $I/O_0 – I/O_7$
- 1 linea di alimentazione
- 1 linea di massa
- una linea CS di abilitazione (attiva bassa)
- una linea di scrittura WE (attiva bassa)
- una linea di lettura OE (attiva bassa)

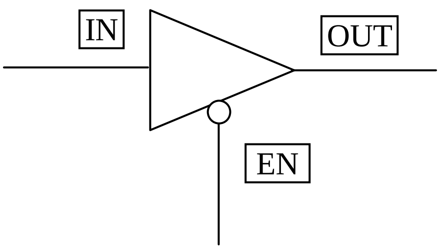
3. PORTE DI INGRESSO/USCITA (I/O)

3.1. Generalità

Le porte di Ingresso/Uscita vengono così chiamate perché consentono un accesso disciplinato al µP da parte di dati provenienti dal mondo esterno o ad esso diretti. Sono costituite da una parte hardware e da una parte software.
La parte hardware è rappresentata da un connettore e da uno o più circuiti integrati (C.I.) che fisicamente consentono o bloccano il passaggio dei dati.
La parte software, invece, è rappresentata da un piccolo programma (driver), specifico per la porta utilizzata, che consente di dare i giusti comandi all'hardware per trasferire i dati.
Le porte che noi utilizzeremo nel prosieguo saranno di tipo molto semplice, per cui non potremo parlare di veri e propri driver ma di semplici istruzioni di controllo. Tali porte costituiscono i circuiti di interfaccia

3.2. Porta di ingresso

Il C.I. che controlla la porta di ingresso è costituita da un banco di Three State.
Tali circuiti vengono così chiamati perché, a differenza delle classiche porte logiche che in uscita presentano i soli stati alto "H" e basso "L", danno origine ad un "terzo stato" che viene denominato ***stato di alta impedenza*** e che sinteticamente viene rappresentato con la sigla "***Hi Z***"
Quando l'uscita di questo dispositivo si trova in uno stato di alta impedenza è come se essa fosse elettricamente scollegata ovvero come se si comportasse come una uscita di un interruttore aperto.
In Fig. 3.1 sono riportati il simbolo logico e la tabella della verità.



EN	IN	OUT
L	L	L
L	H	H
H	X	Hi Z
La X indica uno qualsiasi dei valori L o H. Hi Z indica lo stato di alta impedenza dell'uscita.		

Fig. 3.1

Descriviamone il funzionamento:

- se il piedino di abilitazione EN risulta attivo (basso, nella fig. 3.1) allora l'uscita è uguale all'ingresso; ciò significa che ***è possibile trasferire in uscita il dato presente all'ingresso.***
- se viceversa il piedino di abilitazione EN non è attivo (alto, nella fig. 3.1), allora ***l'uscita si pone in uno stato di alta impedenza (Hi Z), qualunque sia l'ingresso***. ***In tale condizione non è possibile trasferire in uscita il dato presente all'ingresso.***

Da quanto illustrato si comprende che l'impiego di questi dispositivi consente il trasferimento in uscita del dato presente sull'ingresso solo in corrispondenza del livello attivo presente sul piedino di controllo.

Una tipica interfaccia di ingresso è il **C.I. 74LS244** (che nel suo interno contiene 8 circuiti 3-state). In Fig 3.2 sono riportati la tabella di verità, il simbolo logico e lo schema di collegamento.

1G-2G	Xi	Yi
L	L	L
L	H	H
H	X	Hi Z
La X indica uno qualsiasi dei valori L o H. Hi Z indica lo stato di alta impedenza dell'uscita.		

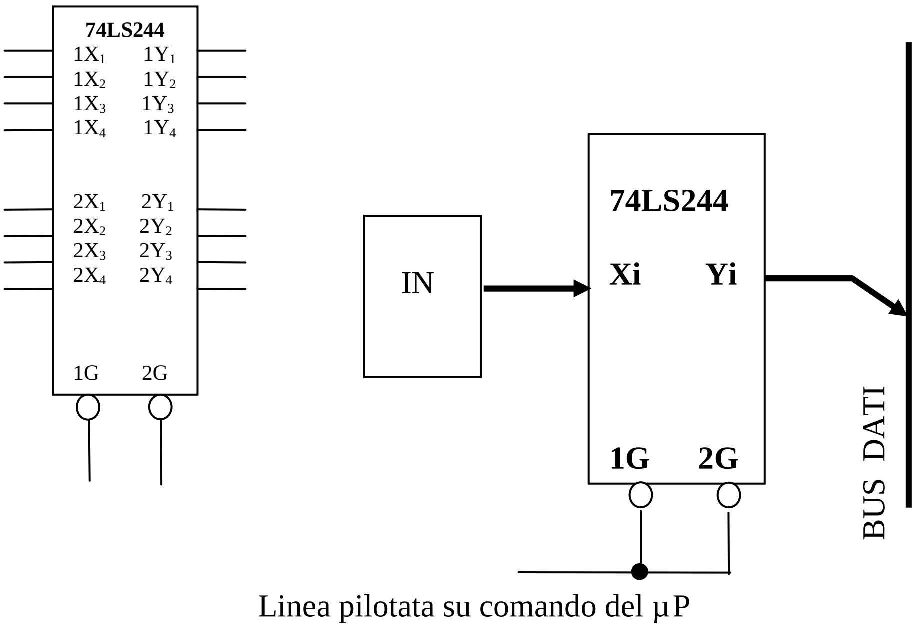


Fig. 3.2

L'ingresso di abilitazione 1G controlla lo stato delle uscite $1Y_1$ – $1Y_4$ mentre l'ingresso di abilitazione 2G controlla lo stato delle uscite $2Y_1$ – $2Y_4$; è evidente che per controllare tutte le uscite, tali due ingressi devono essere collegati e pilotati insieme.
Un apposito circuito di decodifica, che verrà descritto nel seguito, provvede a pilotare, su comando del µP, l'ingresso di abilitazione comune, realizzando il collegamento tra il periferico di IN al BUS DATI.

3.3. Porta di uscita

Il C.I. che consente di memorizzare i dati nella porta di uscita è costituito da un banco di Flip Flop di tipo D. Nella Fig. 3.3 sono riportati il simbolo logico e la tabella della verità per un Flip Flop D elementare.

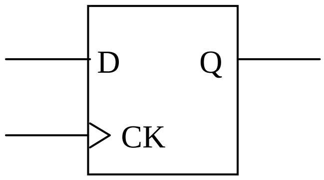
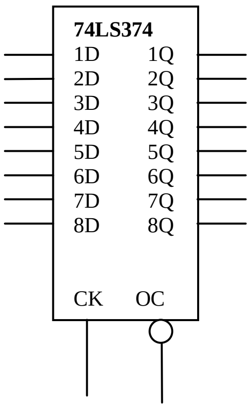


Fig. 3.3

CK	D	Q
X	X	Uscita precedente
↑	H	H
↑	L	L
La X indica uno qualsiasi dei valori L o H. La freccia in su ↑ indica il fronte di salita del clock.		

Descriviamone il funzionamento: quando sull'ingresso di clock (CK) arriva un fronte di salita il livello logico presente sull'ingresso D viene trasferito sull'uscita Q e memorizzato. Eventuali variazioni del livello logico su D non influenzano più l'uscita Q fino a quando non arriva un nuovo fronte di salita sull'ingresso CK. E' importante sottolineare che la memorizzazione del bit in ingresso avviene solo nell'istante corrispondente al fronte di salita del clock: in corrispondenza di tale istante il dato sull'ingresso D deve essere già pronto; passato tale istante, l'ingresso non influenza più l'uscita. Mettendo insieme 8 di questi dispositivi elementari si ha la possibilità di memorizzare un intero byte.
Una tipica interfaccia di uscita è il **C.I. 74LS374** (che nel suo interno contiene 8 flip flop D).
In Fig. 3.4 sono riportati il simbolo logico, la tabella di verità e lo schema di collegamento.



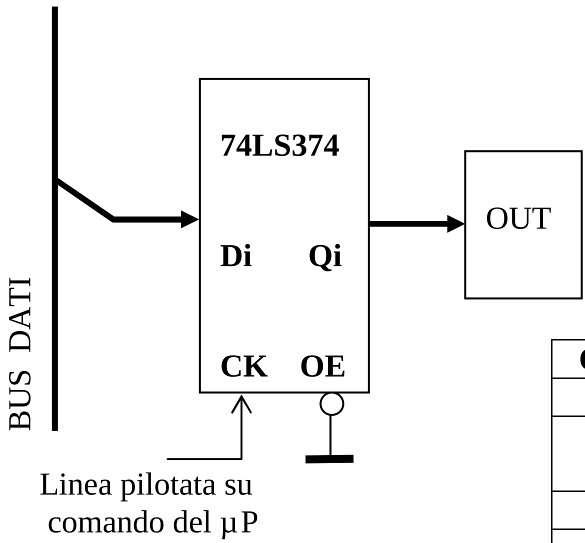


Fig. 3.4

OE	CK	D	Q
H	X	X	Hi Z
L	X	X	Uscita precedente
L	↑	H	H
L	↑	L	L
La X indica uno qualsiasi dei valori L o H. La freccia ↑ indica il fronte di salita del clock. Hi Z indica lo stato di alta impedenza dell'uscita.			

Dalla tabella si nota che il piedino Output Enable (OE) controlla lo stato delle uscite; esso deve essere posto a livello logico basso per consentire la visualizzazione del dato sui piedini di uscita. Il segnale di clock CK, attivato dal µP mediante un apposito circuito di decodifica, consente di memorizzare un nuovo dato sul fronte di salita dello stesso segnale; il segnale di abilitazione per l'uscita OE, essendo sempre attivato, consente il trasferimento del dato dal BUS DATI al periferico di OUT.

3.4. Interfacciamento dei periferici di I/O

I periferici di I/O vengono collegati sul bus dati attraverso le porte di I/O precedentemente descritte, secondo lo schema di principio riportato in fig. 3.5. Per semplificare lo schema sono stati rappresentati un solo 3-state in ingresso (dove ne occorrono 8) ed un solo Flip Flop D in uscita (dove ne occorrono 8).
Come si nota, i periferici di ingresso si collegano al bus attraverso una circuito di ***interfaccia tipo 3-state***; quelli di uscita attraverso un circuito di ***interfaccia tipo Flip-Flop tipo D.***
Si nota anche che i piedini di abilitazione di entrambi i circuiti sono pilotati dal circuito di decodifica indirizzi (Tale circuito verrà esaminato nel capitolo 5°).

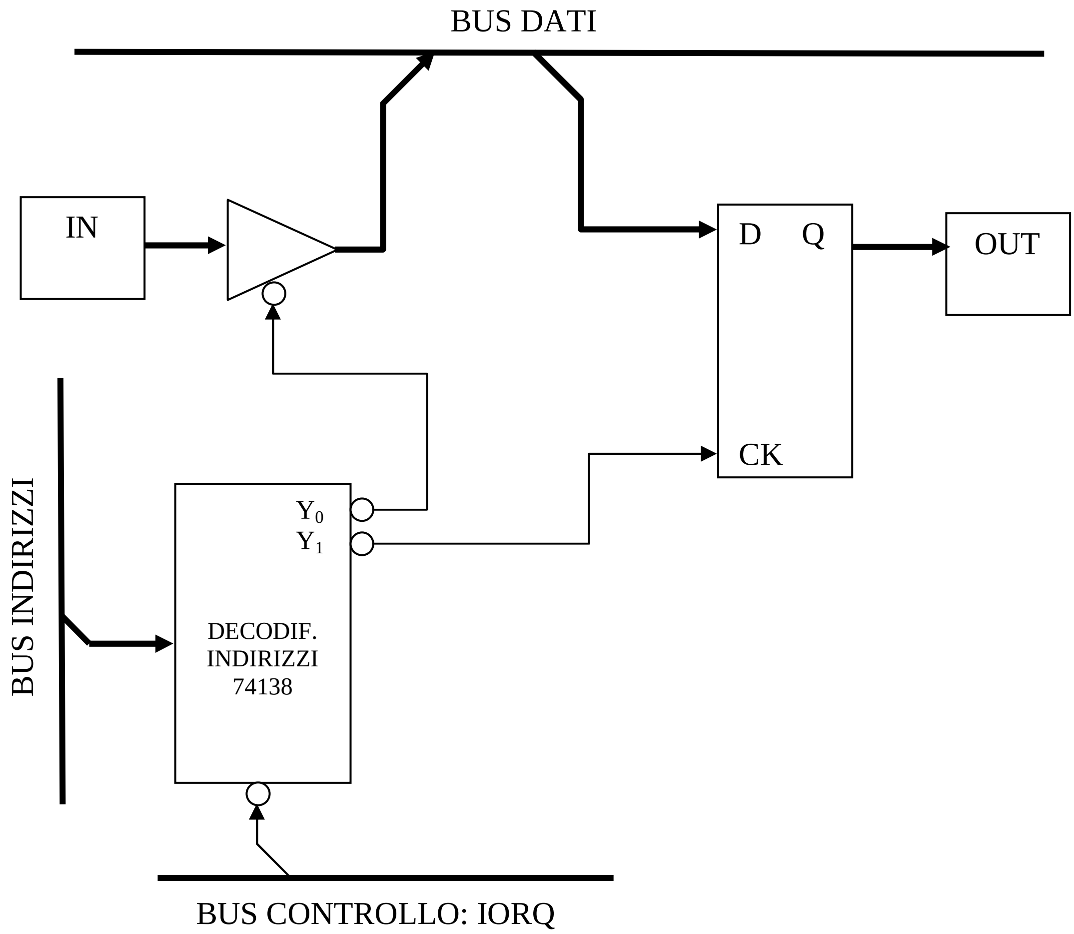


Fig. 3.5

Le problematiche sono completamente diverse a seconda che si tratti di periferici di input (IN) o di output (OUT); per tale motivo esaminiamo separatamente i due casi.

3.4.1. Interfacciamento dei periferici di input

Nell'ipotesi di collegamento diretto sul bus di un periferico di ingresso, i suoi dati risulterebbero ***permanentemente*** presenti sulle linee D_0-D_7 per cui non potrebbero transitare altri dati sullo stesso bus. Per evitare conflitti, ovvero per gestire il bus a divisione di tempo, i periferici di Input vengono collegati ad esso attraverso l'interfaccia di ingresso costituita da un banco di 3-state.
La fig. 3.5 mostra come l'uscita Y_0 del circuito di decodifica comanda l'ingresso di abilitazione del banco di 3-state. Normalmente Y_0 è a livello alto e dunque normalmente il 3-state è nello stato di alta impedenza: ***anche se il periferico di IN , attraverso la sua interfaccia, è collegato fisicamente al bus dati, risulterà elettricamente non collegato e quindi i dati non possono trasferirsi sul bus.***
Quando, durante lo svolgimento del programma, la CPU esegue una istruzione di IN, per i motivi che saranno illustrati nel capitolo 5°, sulla uscita Y_0 si presenta un breve impulso negativo. Tale impulso manda basso il livello logico sul piedino di abilitazione del 3-state, consentendo in tal modo il trasferimento sul bus dati del dato presente sul periferico di ingresso.
In sostanza l'interfaccia di ingresso consente il trasferimento dei dati sul bus solo in corrispondenza della specifica istruzione software di input.

3.4.2. Interfacciamento dei periferici di output

Per i periferici di uscita il problema è il seguente.
Quando, durante lo svolgimento del programma, la CPU deve eseguire una scrittura di un dato sul periferico di OUT, questo dato risulta presente sul bus dati solo per un brevissimo istante. Per questo motivo è necessaria una interfaccia di uscita capace di memorizzare tale dato: si utilizzano Flip Flop D attivi sui fronti e l'abilitazione avviene attraverso l'ingresso di CK.
Sempre con riferimento alla fig. 3.5, si vede come l'uscita Y_1 del circuito di decodifica comanda l'ingresso di CK del flip flop. Normalmente Y_1 è a livello alto e di conseguenza normalmente il F.F. si trova in uno stato di memorizzazione: ***in queste condizioni, qualsiasi dato transiti sul bus dati , ovvero qualsiasi sia il dato presente sull'ingresso D, l'uscita Q resta memorizzata al valore precedente e quindi il dato sul periferico di OUT non si modifica.***
Quando, durante lo svolgimento del programma, la CPU deve eseguire una istruzione di OUT, sulla uscita Y_1 si presenta un breve impulso negativo che viene inviato sull'ingresso di CK del Flip Flop; in corrispondenza del fronte attivo viene acquisito e memorizzato il dato presente sul bus dati.
In sostanza l'interfaccia di uscita consente la cattura dei dati presenti sul bus solo in corrispondenza della specifica istruzione software di output.

4. IL MICROPROCESSORE

4.1. Architettura e principio di funzionamento della CPU

La descrizione che segue si riferisce al dispositivo commerciale Z80; tale descrizione è stata condotta volutamente in termini generali e semplificati in quanto, tenuto conto delle finalità didattiche del Corso, più che lo studio specifico del microprocessore se ne vuole analizzare la logica complessiva.
La fig. 4.1 rappresenta la struttura interna, o come spesso viene chiamata, l'architettura interna di un microprocessore ad 8 bit quale è lo Z80.
Esso è essenzialmente composto da:

- una ***ALU***, Unità Logico-Aritmetica, che esegue tutte le operazioni aritmetiche e logiche;
- da una ***Unità di Controllo***, che genera tutti i segnali necessari per il funzionamento della CPU e del sistema ;
- da una serie di ***registri***. Dal punto di vista didattico è utile dividere i registri interni in due gruppi: quelli che sono coinvolti nel principio di funzionamento della CPU (*registri speciali*) e quelli che, invece, vengono direttamente utilizzati dal progettista in fase di programmazione (*registri di uso generale).*

Esaminiamo in breve il principio di funzionamento della CPU menzionando i relativi registri coinvolti.
Al momento dell'accensione del microprocessore, un apposito circuito di reset inizializza la CPU; tra le varie operazioni di inizializzazione viene eseguita l'operazione di azzeramento del registro ***Contatore di Programma*** (Program Counter = ***PC***) che pertanto contiene il primo indirizzo di locazione di memoria ROM (come sarà spiegato in seguito questo primo indirizzo è, in esadecimale, 0000H) dove è contenuto il codice operativo della prima istruzione da eseguire.
Attraverso il bus interno, il PC invia il primo indirizzo al ***Registro Indirizzi di Memoria*** (Memory Address Register = ***MAR***) che lo trasferisce sul bus indirizzi esterno della CPU. In questo modo si accede alla prima locazione ROM e ***si preleva il codice operativo della prima istruzione.***
Questo dato binario, attraverso il bus dati esterno, giunge al ***Registro Istruzioni*** (Instruction Register = ***IR***) che lo trasferisce al ***Registro di Decodifica delle Istruzioni.*** Questo circuito è una piccola ROM interna nella quale sono memorizzati, per ogni codice operativo che identifica una delle istruzioni eseguibili, un insieme di bit che determinano i segnali di controllo in uscita dalla ***Unità di Controllo***.
I segnali di controllo generati dalla Unità di Controllo sono diretti:

- verso l'esterno, attraverso il bus controllo e sono segnali di attivazione per i vari periferici eventualmente interessati all'esecuzione della istruzione in corso;
- verso l'interno, attraverso linee di controllo genericamente rappresentate dalla freccia "C", per l'attivazione di elementi interni alla CPU interessati all'esecuzione dell'istruzione in corso.

L'attivazione dei segnali di controllo consente l'esecuzione dell'istruzione in corso.

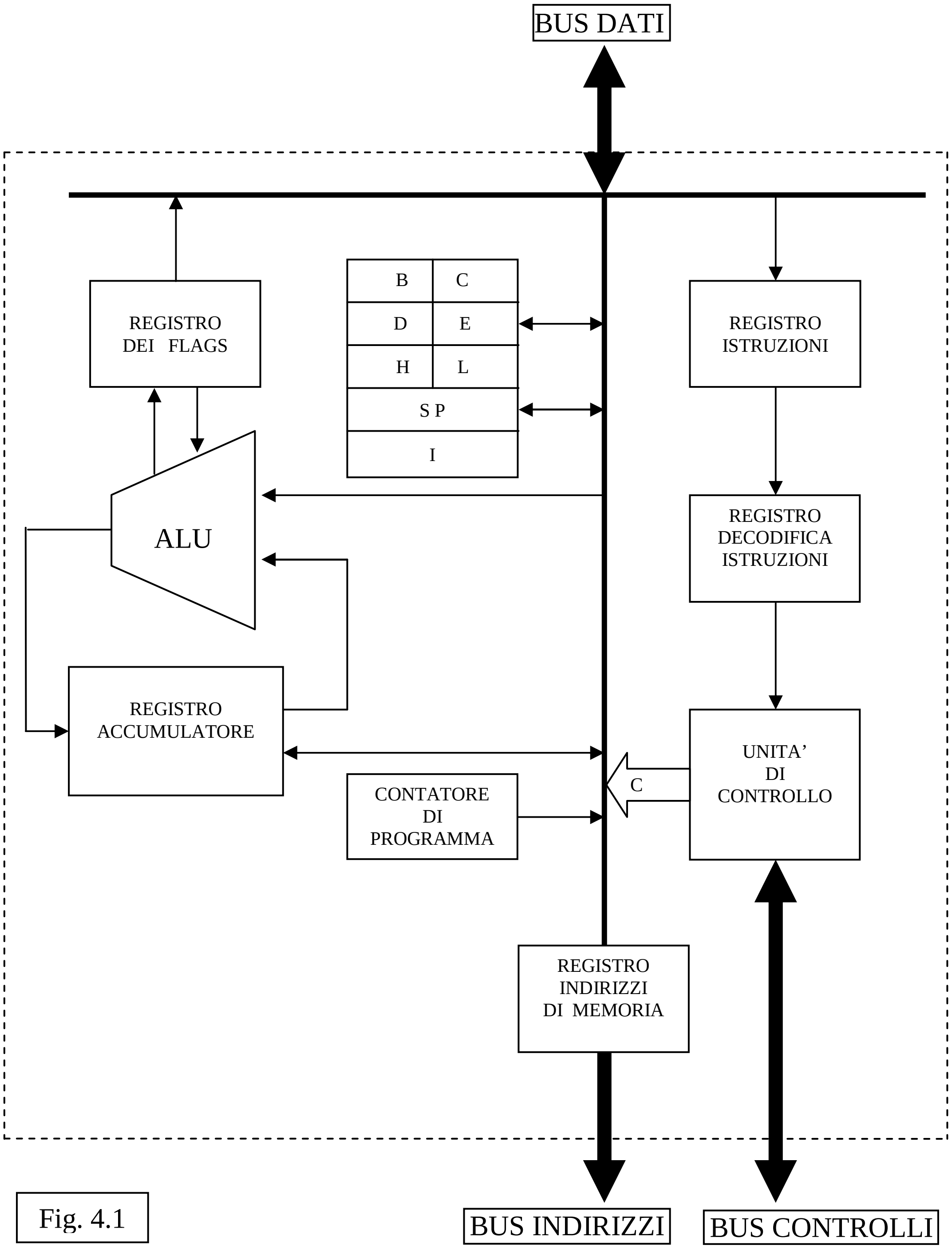


Fig. 4.1

Dopo il prelievo del codice operativo relativo alla prima istruzione, il Contatore di Programma (PC) incrementa automaticamente il suo contenuto e pertanto contiene l’indirizzo della seconda locazione di memoria. A questo punto possono accadere due cose:

a) l’istruzione va completata con un ulteriore accesso alla memoria ROM per il prelievo dell’operando (o degli operandi)
b) oppure l’istruzione è completa perché costituita dal solo codice operativo, per cui non occorrono ulteriori accessi alla memoria ROM

In ogni caso, con l’istruzione completamente prelevata e decodificata, la CPU passa alla fase di esecuzione della stessa. Dopo la prima, viene prelevata ed eseguita la seconda istruzione, e così di seguito. In definitiva il funzionamento del microprocessore non è altro che la ciclica, rapidissima ripetizione della sequenza:

PRELIEVO → DECODIFICA → ESECUZIONE

delle istruzioni che costituiscono il programma.

4.2. Struttura a bus

L'esame della figura 4.1 ci consente di evidenziare la modalità con la quale avviene lo scambio di dati tra i vari blocchi all'interno della CPU e lo scambio di informazioni tra CPU ed i blocchi funzionali esterni ROM, RAM, IN e OUT.
I dati all'interno della CPU transitano attraverso un'unica strada comune a tutti i blocchi, che prende il nome di ***bus dati interno***.
Per quanto riguarda invece lo scambio di informazioni che deve avvenire con i blocchi funzionali esterni ROM, RAM, IN e OUT, il µP provvede a svolgere questa funzione attraverso tre distinti gruppi di linee:

- un gruppo di linee che costituiscono il ***bus dati***
- un gruppo di linee che costituiscono il ***bus indirizzi***
- un gruppo di linee che costituiscono il ***bus controllo***

- Il ***bus indirizzi*** consente alla CPU la ***selezione del dispositivo*** con il quale deve colloquiare. Tale bus è unidirezionale nel senso che è solo la CPU a generare gli indirizzi.
- Il ***bus dati*** consente il ***trasferimento delle istruzioni e dei dati*** tra i singoli blocchi funzionali. E' evidente che tale trasferimento è bidirezionale.
- Il ***bus controlli*** è costituito da una serie di linee la cui attivazione consente la ***gestione delle operazioni*** svolte dalla CPU così come nel dettaglio andremo a specificare successivamente.

Alla luce di quanto esposto, lo schema a blocchi riportato nella Fig. 1.1 del Cap.1 può ora essere schematizzato in modo più esplicito secondo la Fig. 4.2: ***il sistema a µP viene realizzato mettendo in comune, per tutti i blocchi funzionali presenti, le linee dati, indirizzi e controlli.*** Tale struttura, tipica dei sistemi a µP, prende il nome di ***struttura a bus***.
Quella però presentata in Fig. 4.2 è ancora una rappresentazione incompleta; in un capitolo successivo se ne darà una definitiva rappresentazione, semplice ma completa.

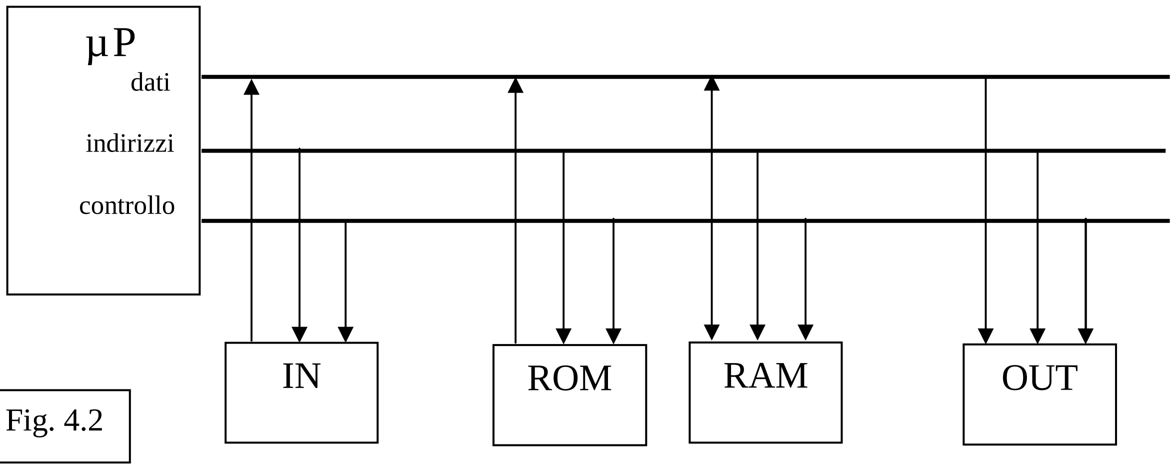


Fig. 4.2

Affinché in un ***sistema organizzato a bus indirizzi comune*** ci sia da parte della CPU la possibilità di selezionare uno dei blocchi funzionali con il quale colloquiare senza generare alcuna ambiguità, ***ad ogni periferico di I/O ed ad ogni locazione di memoria ROM e RAM viene assegnato un preciso ed univo codice (indirizzo) che renda il dispositivo riconoscibile tra tutti gli altri.***
L'hardware che consente tale univoco riconoscimento è un opportuno circuito, chiamato ***circuito di selezione (o di decodifica)***; esso è tipicamente realizzato impiegando decodificatori 74138. Lo studio della selezione delle memorie e dei periferici di I/O verrà affrontato in un capitolo successivo.
Affinché in un ***sistema organizzato a bus dati comune*** non ci sia sovrapposizione di dati in transito, ovvero, come generalmente si dice, non ci sia conflitto, il bus deve essere gestito "***a divisione di tempo***".

La CPU, eseguendo apposite istruzioni, definisce la "sorgente" del dato e la "destinazione" dello stesso; nel breve tempo occorrente per l'esecuzione del trasferimento uno solo tra tutti i dispositivi fisicamente collegati sul bus viene abilitato a trasmettere il dato mentre gli altri devono risultare elettricamente disconnessi.
Per i periferici di I/O l'hardware che consente tale funzionamento è tipicamente realizzato impiegando appositi ***circuiti di interfaccia***; viceversa per le memorie non è richiesto alcun circuito di interfaccia in quanto, come illustrato nel capitolo 2, quando non sono coinvolte nelle operazioni di lettura (ROM) e scrittura - lettura (RAM) esse ***presentano le proprie linee dati in uno stato di alta impedenza.***
Lo studio dell'interfacciamento delle memorie e dei periferici di I/O verrà affrontato in un capitolo successivo.

4.3.Registri interni utilizzati in fase di programmazione

E' consuetudine rappresentare i registri interni del µP come riportato in fig. 4.2.

Registri di uso generale	
A	**F**
B	**C**
D	**E**
H	**L**

Registri di uso speciale	
I	**R**
IX	
IY	
PC	
SP	

Fig. 4.2

I registri costituiscono una ***piccola area di memoria RAM interna alla CPU***. Alcuni di essi possono essere usati ***singolarmente***, ed in tal caso sono registri capaci di memorizzare dati ad ***8 bit***, o ***a coppie*** formando registri in grado di memorizzare informazioni a ***16 bit,*** come ad esempio gli indirizzi delle locazioni di memoria. Questi registri sono quelli denominati **A, F, B, C, D, E, F, H, L, I, R**. Va detto però che i registri F ed R sono registri di sistema gestiti direttamente dal µP. In alcuni casi (quando si usano le istruzioni di PUSH e POP illustrate in un capitolo successivo) si utilizza la coppia AF ma è bene precisare che in questa particolare coppia sono presenti gli 8 bit memorizzati in A ed i flags memorizzati in F.
Il registro **A (A**ccumulatore) è un registro ad 8 bit ma, a differenza degli altri, è un registro preferenziale nel senso che, tra l'altro:

- per tutte le operazioni aritmetiche e logiche che esegue l'ALU uno dei due operandi è contenuto in esso.
- tutti i risultati delle operazioni aritmetiche e logiche eseguite dalla ALU vengono automaticamente memorizzati in esso.
- tutti gli scambi tra CPU e periferici di I/O avvengono attraverso esso.

Il registro **F**, chiamato registro dei **F**lags o registro di stato è di fondamentale importanza in quanto consente, come verrà ampiamente spiegato nel capitolo successivo dedicato alla programmazione, l'esecuzione dei ***salti condizionati*** nell'ambito di un programma.
Nello strutturare un programma è essenziale avere la possibilità di poter seguire due strade, tra loro alternative, a seconda del verificarsi o meno di una determinata condizione. Le condizioni in base alle quali si determina o meno un salto (salto condizionato), sono segnalate da alcuni bit, chiamati flag, contenuti in questo particolare registro.
Esso contiene, secondo quanto rappresentato in fig. 4.3, i seguenti 6 flags:

- S = flag di segno;
- Z = flag di zero
- H = flag di half carry
- p/v = flag di parità/overflow

- N = flag di addizione/sottrazione
- C = flag di carry

S	**Z**	X	**H**	X	**P/V**	**N**	**C**
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

Fig. 4.3

Si noti che i bit b_3 e b_5 non sono utilizzati mentre gli altri sei bit, rappresentativi di altrettanti flags, vengono settati, ovvero portati ad 1 o resettati, ovvero mantenuti a 0 in base alle operazioni eseguite dalla CPU. Nelle tabelle delle istruzioni dei microprocessori, dove sono riportate tutte le possibili istruzioni eseguibili, viene indicato, in corrispondenza di ognuna di esse, l'eventuale modifica dei flags interessati. Esula dai nostri scopi la descrizione dettagliata di tutti i flag; in relazione ai salti condizionati risultano particolarmente importanti i seguenti flags.

- **Flag di Zero**. Segnala, posizionandosi ad 1, il risultato = 0 di una operazione aritmetica o logica. Tale flag si posiziona ad 1 anche nelle istruzioni utilizzate per conoscere il livello logico di un bit, quando il bit testato è basso.

- **Flag di Carry.** Segnala, posizionandosi ad 1, la presenza di un riporto nel caso di una operazione di somma e la presenza di un prestito nel caso di una operazione di sottrazione. Nelle istruzioni di confronto si setta nel caso in cui il dato contenuto nell'accumulatore risulta minore od uguale rispetto all'altro dato con il quale viene confrontato. Nelle istruzioni di rotazione e di scorrimento, assume il valore del bit b7 o del bit b0 a seconda che le rotazioni o gli scorrimenti avvengano rispettivamente verso sinistra o destra.

- **Flag di Segno.** Assume lo stesso valore del bit più significativo di un risultato che normalmente è contenuto nell'accumulatore.
  Siccome nella rappresentazione dei numeri relativi in complemento a due il bit b7 rappresenta un numero positivo se è = 0 o un numero negativo se è = 1, tale flag viene denominato di segno. Spesso questo flag genera confusione: S indica un risultato positivo o negativo solo nel caso in cui il byte contenuto in A rappresenti un risultato numerico espresso da numeri relativi. Se si esegue una somma tra due numeri ad 8 bit e b7 è alto, il flag S si setta ma certamente il suo valore alto non può indicare un risultato negativo.

Il registro **I** (**I**nterrupt register) serve per gestire le interruzioni di modo 2; il suo ruolo verrà chiarito quando si parlerà di interruzioni. Nell'ambito di questo paragrafo precisiamo soltanto che esso serve per memorizzare 8 bit che rappresentano la parte alta di un indirizzo di memoria RAM (tabella vettori) dove si trova un dato a 16 bit che definisce l'indirizzo della subroutine di servizio dell'interruzione che viene caricato nel registro PC.
Il registro **R** (**R**efresh), infine, svolge le funzione di contatore e serve per il rinfresco di memorie dinamiche eventualmente presenti nel sistema considerato; questa sua funzione è un po' simile a quella svolta dal PC per le istruzioni.
Ci sono poi registri che possono essere utilizzati esclusivamente per dati a 16 bit. Questi registri sono denominati **IX, IY, PC, SP.**
I registri **IX** e **IY** si differenziano dagli altri registri a 16 bit per la possibilità di essere utilizzati in combinazione con uno spiazzamento (si vedrà negli esempi pratici cosa questo significhi).
Il registro **PC** (**P**rogram **C**ounter) è un registro di sistema gestito direttamente dal µP; come abbiamo già avuto modo di dire, esso tiene il conteggio delle istruzioni eseguite e quindi indica al µP qual è l'indirizzo dell'istruzione che deve essere eseguita in un dato momento.
Il registro **SP (S**tack **P**ointer), similmente al registro PC, serve per contenere un indirizzo di memoria; in particolare serve per indirizzare l'ultima parte di memoria RAM che viene denominata ***area di stack***. Tale area RAM, come verrà spiegato successivamente, consente il temporaneo salvataggio del contenuto di coppie di registri ed è indispensabile per la gestione dei sottoprogrammi e delle interruzioni.

Evidenziamo le differenze tra PC ed SP:

- il PC contiene indirizzi di memoria ROM mentre lo SP contiene indirizzi di memoria RAM;
- il PC, all'atto dell'avvio della CPU, viene automaticamente inizializzato con il primo indirizzo ROM mentre lo SP viene inizializzato dal progettista del software con un indirizzo RAM che definisce il "fondo" dell'area di stack;
- il PC, per ogni istruzione eseguita, automaticamente incrementa il proprio indirizzo mentre lo SP, per ogni salvataggio eseguito decrementa il proprio indirizzo. Questo suo modo di funzionare fa sì che l'area RAM interessata al salvataggio temporaneo di dati, si comporti in modo LIFO (Last In - First Out) ovvero come una particolare memoria dove l'ultimo dato salvato è il primo ad essere letto.

4.4. Il microprocessore come Circuito Integrato

In questo paragrafo vogliamo studiare il µP come blocco funzionale, attraverso l'esame dei segnali presenti sulle sue linee. Nella Fig. 4.4 sono stati riportati solo i segnali che incontreremo nel nostro studio; altri segnali, più o meno complessi che non studieremo, non sono stati considerati.

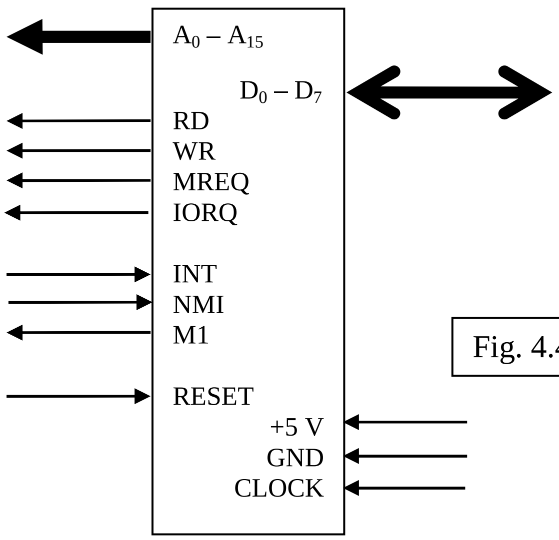


Fig. 4.4

- ***8 linee dati $D_0 – D_7$*** che costituiscono il ***bus dati***.

Queste otto linee rappresentano la "strada" attraverso la quale passano tutte le informazioni. Queste informazioni sono ad 8 bit in quanto microprocessori tipo Z80 e similari sono ad otto bit (esistono µP i cui dati sono costituiti da parole più lunghe per cui si hanno µP più sofisticati a 16 bit, 32 bit e 64 bit).

I dati che transitano sul bus dati sono quelli che vanno verso la memoria (***operazione di scrittura in memoria → dati in uscita dal µP***), che provengono dalla memoria (***operazione di lettura dalla memoria → dati in ingresso al µP***), quelli che provengono da una periferica di ingresso (***operazione di lettura da un periferico di ingresso → dati in ingresso al µP***) ed infine quelli che vanno verso una periferica di uscita (***operazione di scrittura in un periferico di uscita → dati in uscita dal µP***). ***Il bus dati è bidirezionale***.

- ***16 linee di indirizzo $A_0 – A_{15}$*** che costituiscono il ***bus indirizzi***.

Per quanto si è detto nel Cap. 2 sulla corrispondenza tra numero di linee di indirizzo e capacità di memoria, è facile calcolare che con 16 linee di indirizzo si possono selezionare 2^{16} locazioni di memoria, pari ad una capacità di memoria di 64 K; siccome stiamo trattando µP ad 8 bit, la massima capacità di memoria indirizzabile da parte di un µP con bus a 16 linee sarà pari a 64 Kbyte.

Per indirizzare le locazioni di memoria si usano tutte le linee $A_0 – A_{15}$ del bus indirizzi.

Come ogni singola locazione di memoria viene identificata con il suo indirizzo, specificato attraverso i 16 bit del bus indirizzi, anche i periferici di ingresso e di uscita vengono identificati con un indirizzo. ***Per indirizzare i dispositivi di I/O si usano le 8 linee meno significative $A_0 – A_7$ del bus.*** In ogni caso, o che si tratti di memorie o che si tratti di periferici, l'indirizzo è sempre una informazione in uscita dal µP per cui ***il bus indirizzi è unidirezionale con verso in uscita.***

- ***8 linee di controllo*** che costituiscono il ***bus controllo***.

Tutte le linee di controllo risultano attive basse. Descriviamo nel modo più semplice possibile il ruolo che ognuna di queste linee svolge nell'ambito del funzionamento del sistema controllato dalla CPU.

- ***M1*** (**M**achine cycle one = primo ciclo macchina): è una linea di controllo ***in uscita dal µP***. Questa linea di controllo diventa attiva in corrispondenza di ogni ciclo di lettura del codice operativo. Inoltre, quando il periferico che vuole colloquiare con la CPU segnala tale evenienza mandando basso INT, la CPU risponde segnalando l'accettazione di interruzione attraverso l'attivazione contemporanea dei segnali M1 ed IORQ.

- ***RD*** (**R**ea**D** = lettura): è una linea di controllo ***in uscita dal µP***. Normalmente il livello logico presente su tale linea è alto; quando il µP vuole leggere un dato (dalla memoria o da un periferico di Input), tale linea va bassa.

- ***WR*** (**WR**ite = scrittura): è una linea di controllo ***in uscita dal µP***. Normalmente il livello logico presente su tale linea è alto; quando il µP vuole scrivere un dato (nella memoria o in un periferico di Input), tale linea va bassa.

- ***MREQ*** (**M**emory **REQ**uest = richiesta di memoria): è una linea di controllo ***in uscita dal µP***. Normalmente il livello logico presente su tale linea è alto; quando il µP vuole attivare la decodifica di memoria, per leggere o scrivere un dato, tale linea va bassa.

- ***IORQ*** (**I**nput/**O**utput **R**e**Q**uest = richiesta di ingresso/uscita): è una linea di controllo ***in uscita dal µP***. Normalmente il livello logico presente su tale linea è alto; quando il µP vuole attivare la decodifica di Ingresso/Uscita, per leggere o scrivere un dato, tale linea va bassa.

- ***INT*** (**INT**errupt request = richiesta di interruzione): è una linea di controllo ***in ingresso al µP***. Il periferico di I/O che vuole colloquiare con la CPU invia un segnale che manda bassa tale linea segnalando appunto una richiesta di colloquio; questa linea viene chiamata "richiesta di interruzione" in quanto, se attivata, la CPU "interrompe" il programma che sta svolgendo per scambiare i dati con il periferico richiedente.

- ***NMI*** (**N**on **M**askable **I**nterrupt = interruzione non mascherabile): è una linea di controllo ***in ingresso al µP***. E' una linea di controllo che svolge una funzione simile a quella descritta in precedenza. La differenza tra interruzioni mascherabili e non mascherabili verrà illustrata quando si parlerà di interruzioni.

- ***RESET*** (azzeramento): è una linea di controllo ***in ingresso al µP***. La sua attivazione comanda l'inizializzazione della CPU. Tra le varie operazioni citiamo l'azzeramento del Contatore di Programma.

- ***HALT*** (stato di arresto): è una linea di controllo ***in uscita dal µP***. Quando la CPU esegue una istruzione di HALT il Contatore di Programma arresta il suo incremento e l'esecuzione del programma viene interrotto; questa condizione viene segnalata con l'attivazione della linea in esame. Quando la CPU si trova nello stato di arresto resta in attesa di un segnale di interruzione.

- ***BUSRQ*** (**BUS** **R**e**Q**uest = richiesta del bus): è una linea di controllo ***in ingresso al μP***. Quando un dispositivo deve prendere il controllo dei bus, comunica tale evenienza alla CPU mandando bassa tale linea; la CPU si "scollega" elettricamente dai bus cedendone il controllo al dispositivo richiedente.

- ***BUSAK*** (**BUS** **A**c**K**nowledge = riconoscimento richiesta controllo dei bus): è una linea di controllo ***in uscita dal μP***. Attivando questa linea, la CPU conferma al dispositivo richiedente che può prendere il controllo dei bus.

- ***RFSH*** (**R**e**F**re**SH =** Rinfresco): è una linea di controllo ***in uscita dal μP***. Quando questa linea è attiva e contemporaneamente risulta attiva MREQ, la CPU utilizza i sette bit A0 – A6 come indirizzo per il rinfresco delle DRAM eventualmente presenti nel sistema.

- ***WAIT*** (attesa): è una linea di controllo ***in ingresso al μP***. Attraverso l'attivazione di questa linea, la memoria o un periferico di I/O segnala alla CPU che non è pronto per trasferire i suoi dati.

Si noti che le 4 linee MREQ, IORQ, RD, WR sono coordinate fra loro, come si evince dal prospetto seguente

- ***Lettura da memoria: vanno basse le linee MREQ e RD***
- ***Scrittura in memoria: vanno basse le linee MREQ e WR***
- ***Lettura da un periferico di Input: vanno basse le linee IORQ e RD***
- ***Scrittura in un periferico di Out: vanno basse le linee IORQ e WR***

5. ARCHITETTURA DI UNA SCHEDA A MICROPROCESSORE

5.1 Architettura della scheda

A questo punto siamo in grado di dare un'occhiata a come è organizzata una scheda che sfrutti un microprocessore Z80 per svolgere i compiti assegnati.
Nella Fig. 5.1 è rappresentata una delle soluzioni più semplici che possiamo avere.

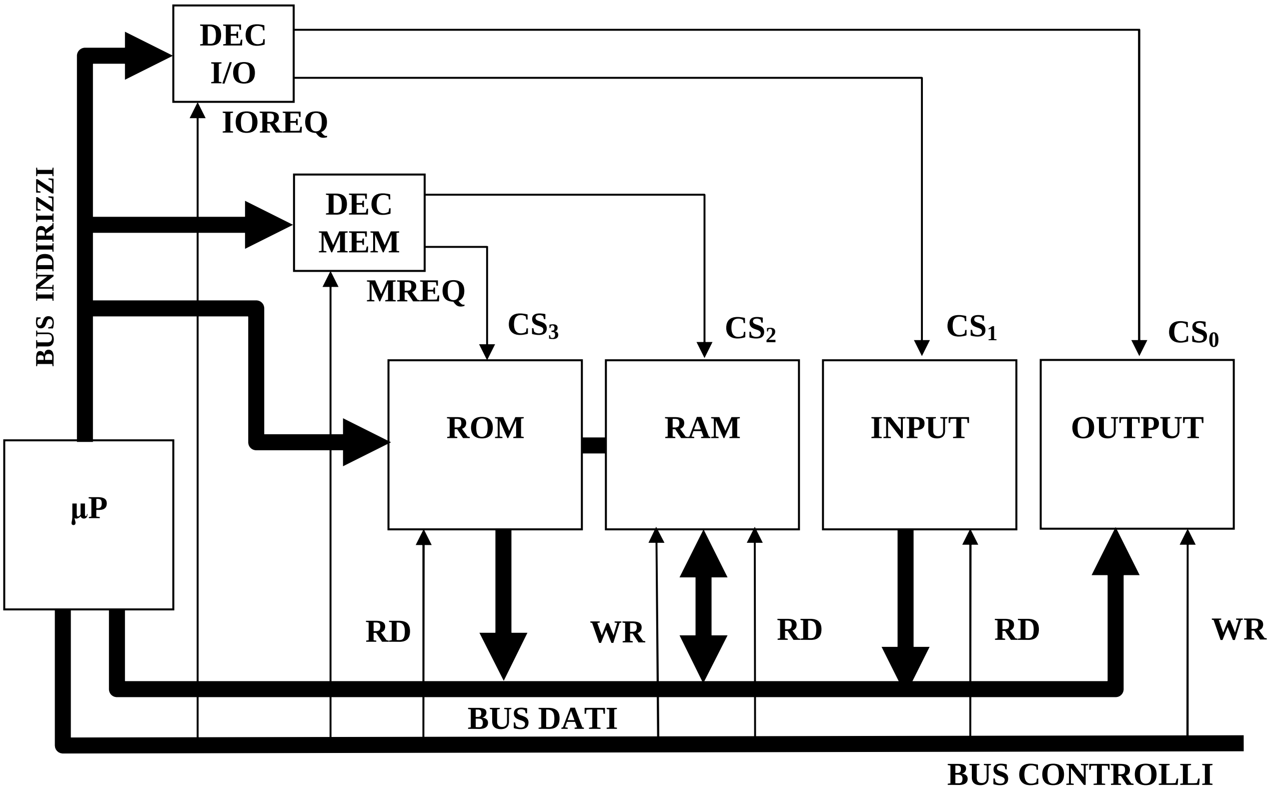


Fig. 5.1

Essa prevede:

a) il µP
b) una memoria EPROM per le istruzioni del programma. Deve per forza trattarsi di una EPROM dato che allo spegnimento del sistema il programma deve essere conservato per essere impiegato successivamente.
c) una memoria RAM per i dati da memorizzare. Deve trattarsi di una RAM perché la EPROM è di sola lettura e quindi in essa non può essere scritto niente.
d) una porta di ingresso, costituita da un banco di Three State
e) una porta di uscita, costituita da un banco di Flip Flop D
f) due decodifiche, una per i dispositivi di memoria e l'altra per le porte di I/O. La decodifica ha il compito di ricevere dal µP il codice identificativo (indirizzo) del dispositivo con il quale il µP vuole colloquiare (locazione di memoria, porta di ingresso o porta di uscita), riconoscere tale codice ed attivare solo il dispositivo interessato. Le due decodifiche, oltre a ricevere l'indirizzo del dispositivo da selezionare, devono essere abilitate, singolarmente, dai segnali di controllo MREQ (**M**emory **REQ**uest) oppure IOREQ (**I**nput/**O**utput **REQ**uest). Questi due segnali di controllo sono attivi bassi e non potranno mai essere attivati contemporaneamente. Le linee tramite le quali la decodifica attiva e disattiva i dispositivi vengono indicate con CS_i (**C**hip **S**elect).

Le informazioni vengono scambiate tra il µP e gli altri dispositivi mediante il BUS DATI, costituito da 8 linee (per µP a 8 bit) che realizzano un collegamento unico tra tutti i dispositivi (µP compreso); su tale collegamento le informazioni viaggiano suddivise in byte. Ovviamente non è possibile lasciare libero accesso al BUS a tutti i dispositivi, perché così si creerebbe solo confusione (conflitto sul BUS). Bisogna allora prevedere un sistema di decodifica che, assegnato un codice particolare (indirizzo) ad ogni dispositivo, consenta di attivare il collegamento tra il µP ed uno solo di tali dispositivi, quello individuato dall'indirizzo emesso dal µP. Le linee sulle quali viene emesso l'indirizzo del dispositivo da selezionare, costituiscono il BUS INDIRIZZI. I due dispositivi che realizzano fisicamente le due decodifiche, una per le memorie e l'altra per le porte di I/O, fanno da tramite tra l'indirizzo emesso dal µP e l'attivazione della locazione di memoria o della porta che è associata a tale indirizzo.
L'organizzazione della scheda è completata da alcune linee di controllo che costituiscono il BUS CONTROLLI. Le linee MREQ e IOREQ, come già detto precedentemente, abilitano la decodifica di memoria (MREQ) e la decodifica di I/O (IOREQ); ricordiamo che in ogni momento può essere attivata una sola di queste due linee. Altre due linee di controllo sono chiamate RD (**R**ea**D**) e WR (**WR**ite): sono attive a livello logico basso e consentono di selezionare, rispettivamente, una operazione di lettura o una operazione di scrittura; anche in questo caso in ogni momento può essere attivata una sola di queste due linee. Si noti che il BUS INDIRIZZI, per le memorie, si divide in due parti: un gruppo di linee va direttamente, e in parallelo, sui piedini di indirizzo dei singoli chip e quindi seleziona le celle con la stessa posizione in tutti i chip; l'altro gruppo seleziona il chip da attivare. Tra le possibili celle selezionabili, una sola viene attivata, quella il cui indirizzo corrisponde pienamente, bit per bit, all'indirizzo inviato dal µP.
Nel paragrafo seguente si esaminerà in modo più esteso il concetto di decodifica.

5.2 I circuiti di decodifica

I ***circuiti di decodifica,*** chiamati anche ***circuiti di selezione***, sono quei circuiti che ***provvedono ad assegnare gli indirizzi*** ad ogni singola locazione di memoria EPROM e RAM presente nel sistema e ad ogni periferico di I/O presente nel sistema.
Ricordiamo che per indirizzare le memorie si utilizza l'intero bus indirizzi mentre per indirizzare i periferici di I/O si utilizza solo la metà meno significativa di tale bus.
Quando si impiegano ***tutte le linee di indirizzo***, ***la decodifica viene detta assoluta;*** quando invece vengono impiegate ***solo in parte le linee di indirizzo***, ***la decodifica viene detta ambigua***.
Per le memorie la decodifica risulta sempre assoluta e questo comporta, come del resto deve essere, che ogni singola locazione presenta un unico indirizzo.
Per i periferici di I/O la decodifica può essere assoluta o ambigua. Nel primo caso il fatto che tutte le linee della parte bassa degli indirizzi vengono impiegate per la selezione consente di avere, come per le memorie, un indirizzo unico per ogni periferico di I/O presente nel sistema. Viceversa nel caso di decodifica ambigua ad ogni periferico di I/O non viene assegnato un unico indirizzo ma un intervallo di indirizzi; qualunque valore compreso in questo intervallo seleziona il periferico in questione. Questa pluralità di indirizzi potrebbe creare qualche confusione nella stesura del programma ma se il progettista software è al corrente di questa scelta hardware, non ha problemi nella corretta indicazione degli indirizzi per la gestione delle istruzioni di IN e OUT. In compenso questa tipologia di decodifica comporta una semplificazione dell'hardware.

5.3 Attivazione di una porta di ingresso

Una porta di ingresso è un dispositivo che preleva un dato (byte) dal mondo esterno attraverso un connettore e lo trasferisce sul BUS DATI; il trasferimento non può avvenire in qualsiasi momento ma deve essere abilitato dal µP.
L'operazione avviene nel modo seguente:

a) Il µP invia alla decodifica di I/O (abilitata mediante l'IORQ) l'indirizzo della porta da selezionare per l'operazione
b) La decodifica, in base al valore dell'indirizzo, attiva una sua linea di uscita (una sola) con la

quale va ad abilitare un banco di Three-State posto lungo il percorso del segnale e che costituisce la vera e propria porta di ingresso (Vedi Par.3.2).

c) Una volta abilitato il Three-State, il segnale può passare dal connettore al BUS DATI e quindi al µP
d) Dopo aver acquisito il dato, il µP disattiva la decodifica (disattivando l'IORQ) e quest'ultima, a sua volta, disattiva il Three-State, scollegando quindi dal connettore il BUS DATI, che torna libero e pronto per la successiva operazione

Nota: nella decodifica di I/O lo Z80 utilizza solo gli 8 bit meno significativi del BUS INDIRIZZI (A7÷A0).

Nella Fig. 5.2 viene illustrato uno schema generale per la lettura di un dato a 8 bit. Il segnale di abilitazione per i Three-State di solito viene chiamato "Chip Select (CS)" oppure "Chip Enable (CE)".

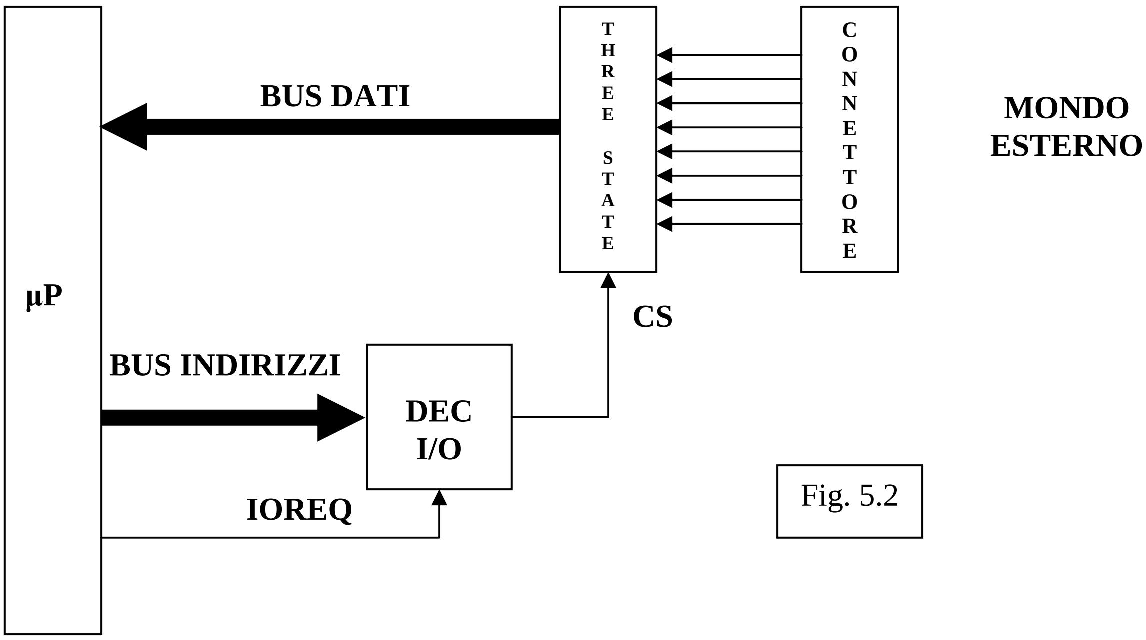


Fig. 5.2

5.4 Attivazione di una porta di uscita

Una porta di uscita è un dispositivo che preleva un dato (byte) dal BUS DATI e lo trasferisce al mondo esterno attraverso un connettore. Anche in questo caso il trasferimento non può avvenire in qualsiasi momento ma deve essere abilitato dal µP; inoltre il dato deve essere subito depositato da qualche parte, in modo da liberare il BUS e consentire quindi altre operazioni di trasferimento (che si contano in termini di milioni al secondo). Per soddisfare entrambe le esigenze, il dato in uscita viene memorizzato in un banco di Flip-Flop di tipo D e lì resta fino a quando non viene sostituito da un nuovo dato (Vedi Par. 3.3).
L'operazione avviene nel modo seguente:

a) Il µP pone il dato da trasmettere sul BUS DATI
b) Invia poi alla decodifica di I/O (abilitata mediante l'IORQ) l'indirizzo della porta da selezionare per l'operazione
c) La decodifica, in base al valore dell'indirizzo, attiva una sua linea di uscita (una sola) con la quale va ad attivare il clock comune a tutti i Flip-Flop e che altro non è se non il Chip Select della porta di uscita.
d) Una volta memorizzato sui Flip-Flop, il dato è disponibile sul connettore. A questo punto il µP disattiva la decodifica (disattivando l'IORQ) e il BUS DATI torna libero e pronto per la successiva operazione

Nella Fig. 5.3 viene illustrato uno schema generale per la scrittura di un dato a 8 bit.

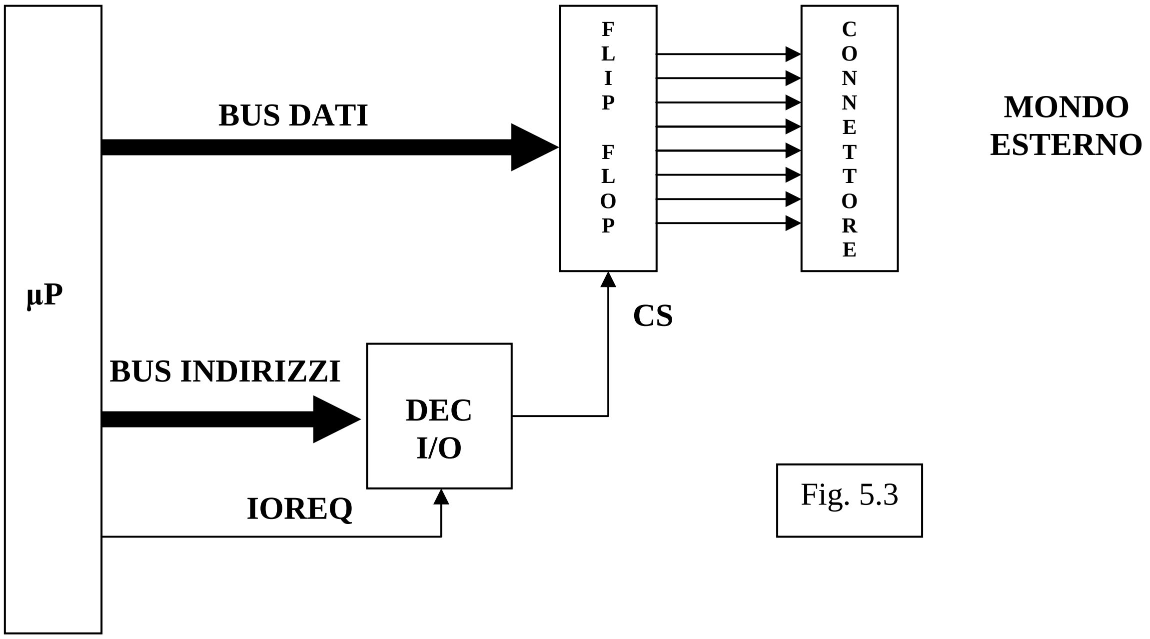

µP
BUS DATI
FLIP FLOP
CONNETTORE
MONDO ESTERNO
CS
BUS INDIRIZZI
DEC I/O
IOREQ

Fig. 5.3

5.5 Attivazione di una porta di ingresso e di una porta di uscita

Nella Fig. 5.4 viene illustrato uno schema che consente di decodificare una porta di ingresso ed una di uscita.

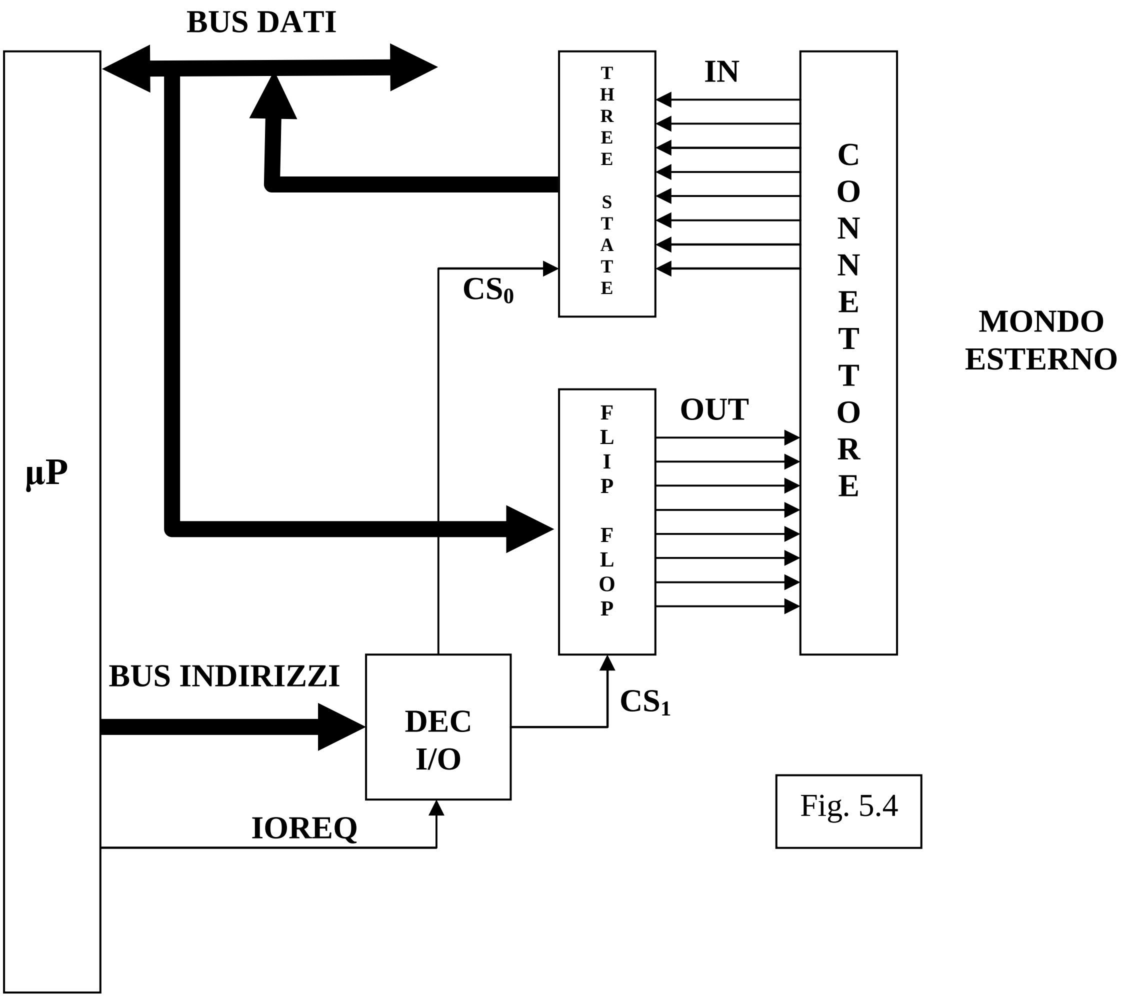


Fig. 5.4

5.6 Esempio di decodifica di porte di I/O

Supponiamo di dover decodificare 5 porte di ingresso con indirizzi, rispettivamente 50H, 51H, 52H, 53H, 54H. La Fig. 5.5 illustra uno schema che risolve il problema.

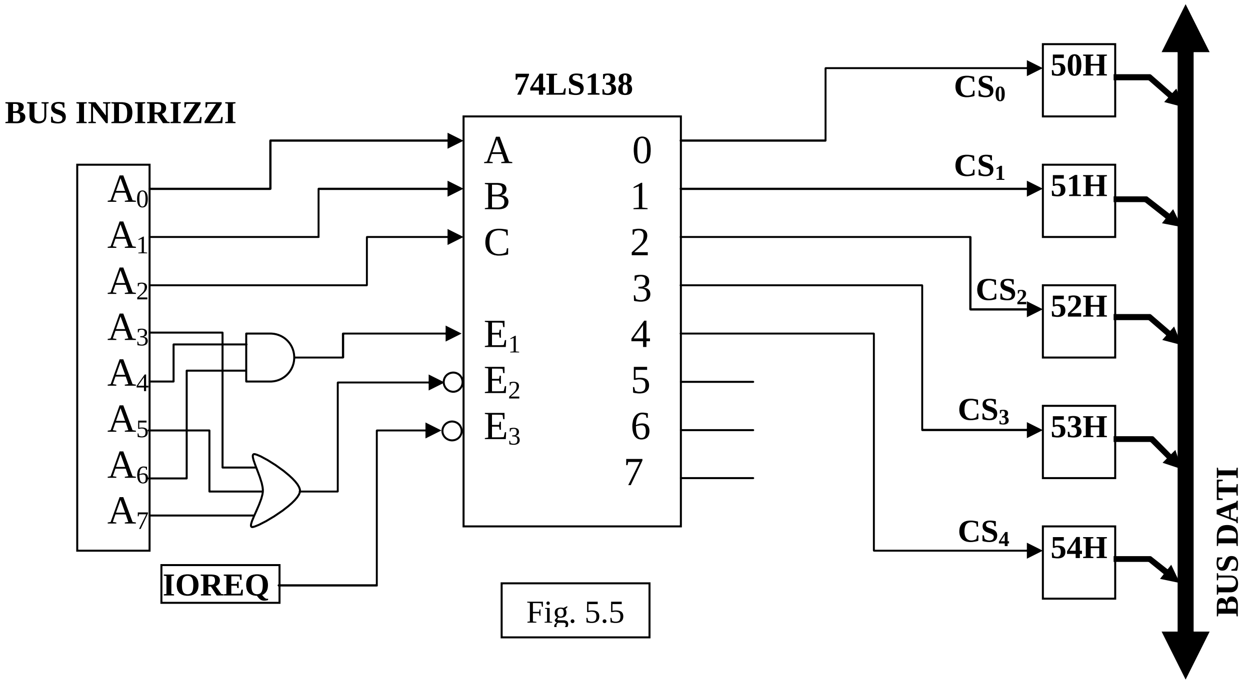


Fig. 5.5

Ricordiamo che il 74LS138 è una decodifica a 3 bit; interpreta la stringa binaria presente sugli ingressi C, B e A (C è l'MSB) e manda a livello logico basso una sola delle 8 uscite, quella corrispondente al valore decimale rappresentato dalla stringa stessa; tutte le altre restano a livello logico alto. Questo accade, però, se sono stati abilitati gli ingressi ausiliari E_1 (alto), E_2 (basso), E_3 (basso); se così non è tutte le uscite sono bloccate a livello logico alto, anche quella selezionata dalla stringa binaria di ingresso.
Per giustificare lo schema di Fig. 5.5, occorre far riferimento alla seguente tabella, nella quale sono stati convertiti in binario gli indirizzi espressi in esadecimale.

HEX	BINARIO							
	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
50	0	1	0	1	0	**0**	**0**	**0**
51	0	1	0	1	0	**0**	**0**	**1**
52	0	1	0	1	0	**0**	**1**	**0**
53	0	1	0	1	0	**0**	**1**	**1**
54	0	1	0	1	0	**1**	**0**	**0**

Delle 8 linee di indirizzo, solo le ultime 3 variano il loro valore logico al variare dell'indirizzo stesso (A_2, A_1, A_0) e quindi contengono in sé l'informazione da decodificare: esse vanno collegate agli ingressi principali, rispettivamente, C, B, A, e provvedono alla selezione della porta. Le altre 5 linee vengono utilizzate per abilitare la decodifica attraverso gli ingressi E_1, E_2, E_3, in modo da avere una decodifica assoluta, vale a dire un unico indirizzo per ciascuna porta. Infatti, se le linee $A_7 \div A_3$ fossero lasciate libere, la selezione della porta di indirizzo 50H, ad esempio, si otterrebbe con una stringa **000** sulle linee A_2, A_1, A_0 e con qualsiasi combinazione di 0 e 1 sulle altre linee non collegate. La porta di indirizzo 50H, quindi, non sarebbe attivata solo dal codice 50H (01010**000**) ma anche dal codice 00H (00000**000**), dal codice 10H (00010**000**), dal codice 20H (00100**000**), e da tanti altri ancora. In totale ogni porta potrebbe essere selezionata con $2^5 = 32$ indirizzi diversi (2^5 perché 5 sarebbero le linee di indirizzo non collegate); tutti questi indirizzi avrebbero in comune il valore dei 3 bit meno significativi (**000**). Lo stesso discorso si può ripetere per le altre porte.

Ora, poiché 5 linee di indirizzo ($A_7 \div A_3$) devono essere collegate a 2 soli ingressi(E1, E_2), si devono utilizzare delle porte logiche scelte in modo opportuno.

a) Le linee di indirizzo con valore logico **0** (A_7, A_5, A_3) vanno all'ingresso di una porta **OR**, la cui uscita è collegata all'ingresso di abilitazione $\mathbf{E_2}$ (attivo basso).
b) Le linee di indirizzo con valore logico **1** (A_6, A_4) vanno all'ingresso di una porta **AND**, la cui uscita è collegata all'ingresso di abilitazione $\mathbf{E_1}$ (attivo alto).
c) Il terzo ingresso di abilitazione $\mathbf{E_3}$ (attivo basso) viene collegato alla linea **IOREQ** proveniente dal microprocessore.

In questo modo le linee A_2, A_1, A_0 provvedono con il loro valore alla selezione della porta, ma se l'operazione di I/O non è richiesta dal microprocessore (IOREQ alto) oppure se almeno una delle linee $A_7 \div A_3$ non ha il giusto valore (**0** per A_7, A_5, A_3 e **1** per A_6, A_4), la decodifica non può operare perché disabilitata.
Chiuso l'esempio, aggiungiamo che è possibile, con un accorgimento, assegnare lo stesso indirizzo a due porte, purché una sia di ingresso e l'altra di uscita. La Fig. 5.6 illustra la variante.

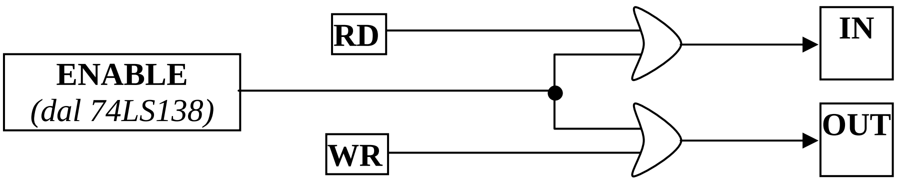


Fig. 5.6

Le due porte OR condizionano il segnale di abilitazione proveniente dal 74LS138, in modo da attivare o l'una o l'altra porta, a seconda di quale segnale di controllo è attivato: RD o WR.

5.7 Decodifica di memoria

Per la decodifica di memoria lo Z80 impiega tutte e 16 le linee del BUS INDIRIZZI. Si possono quindi decodificare fino a $2^{16} = 65536$ indirizzi diversi. Ogni chip di memoria prevede al suo interno una propria decodifica che consente di associare all'indirizzo prelevato dai piedini dell'integrato la corrispondente locazione. Se per necessità i chip sono più di uno, occorrerà anche una decodifica esterna che consenta di selezionare quello in cui è posizionata la locazione che interessa. Occorrerà quindi suddividere il totale delle linee di indirizzo in due gruppi: le linee meno significative vanno a selezionare la locazione dentro il chip; le più significative vanno a selezionare uno solo tra i vari chip, quello appunto dove è ubicata la locazione. L'esempio che segue chiarirà quanto detto.

5.8 Esempio di decodifica di memoria

Progettare la decodifica per un banco di memorie costituito da 4 memorie da 2K.
Nota: è importante che i chip di memoria abbiano tutti la stessa capacità.

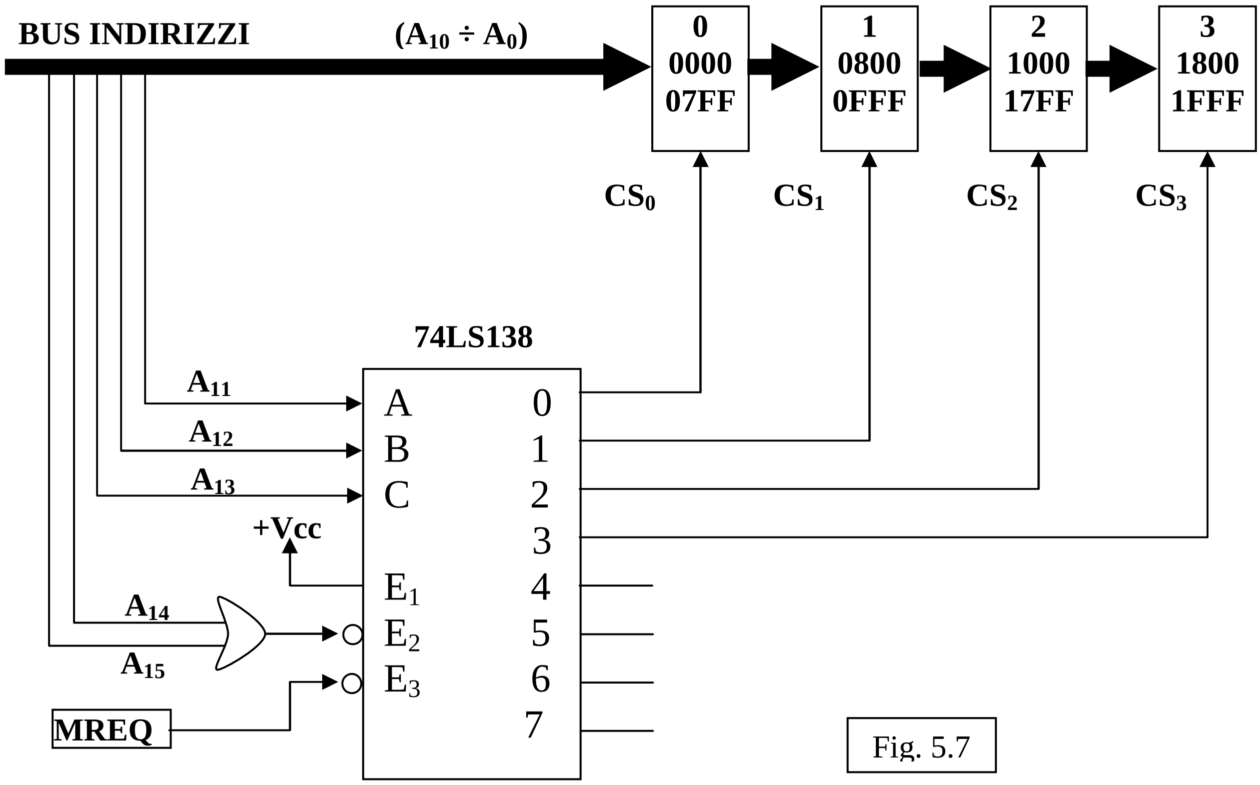


Fig. 5.7

Nella Fig. 5.7, per ogni chip di memoria, viene indicata la numerazione progressiva in esadecimale delle locazioni.
Il totale di 16 linee di indirizzo viene suddiviso in due gruppi. Le prime 11 ($A_{10} \div A_0$) vanno collegate a tutti e 4 i chip di memoria previsti, in parallelo come se si volesse in una sola volta selezionare la stessa locazione in tutti e 4 i chip. Le linee del secondo gruppo ($A_{15} \div A_{11}$) vengono invece utilizzate per individuare, tra i 4, il chip che contiene la locazione interessata; vengono quindi inviate ad un circuito di decodifica esterno, del tutto simile a quello utilizzato per le porte di I/O. Ovviamente, ad attivare la decodifica è questa volta il MREQ.
Per sapere come collegare le linee ($A_{15} \div A_{11}$) alla decodifica esterna, occorre ancora una volta convertire in binario l'indirizzo, espresso in esadecimale, della prima locazione di ogni chip e considerare solo i bit **15, 14, 13, 12, 11**.

CHIP	HEX	BINARIO															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0800	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
2	1000	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
3	1800	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
		Alla decodifica esterna					**Ai chip di memoria**										

I bit che variano al variare della riga (A_{12} e A_{11}) vengono inviati agli ingressi B e A della decodifica. Poiché resta libero l'ingresso C, ad esso viene collegato il bit A_{13}.
Per rendere assoluta la decodifica, i bit A_{15} e A_{14} vengono inviati, attraverso la porta OR, ad uno degli ingressi di abilitazione attivi bassi. All'altro ingresso di abilitazione attivo basso viene inviato il MREQ. L'ultimo ingresso di abilitazione, quello attivo alto, viene attivato collegandolo direttamente all'alimentazione.

5.9 Le temporizzazioni

Nei paragrafi successivi vogliamo mettere in evidenza lo strettissimo legame esistente tra il software che viene svolto dalla CPU, vista come esecutrice di un programma, e le corrispondenti risposte hardware che la stessa CPU, vista come C.I., fornisce verso l'esterno attraverso le sue linee indirizzi, dati e controlli. Le evoluzioni nel tempo che presentano le linee indirizzi, controlli e dati, sono dette temporizzazioni.

5.9.1. Temporizzazione di una memoria

Quanto diremo in questo paragrafo vale, con qualche piccola variante, sia per le EPROM che per le RAM.

5.9.1.1. Lettura da memoria

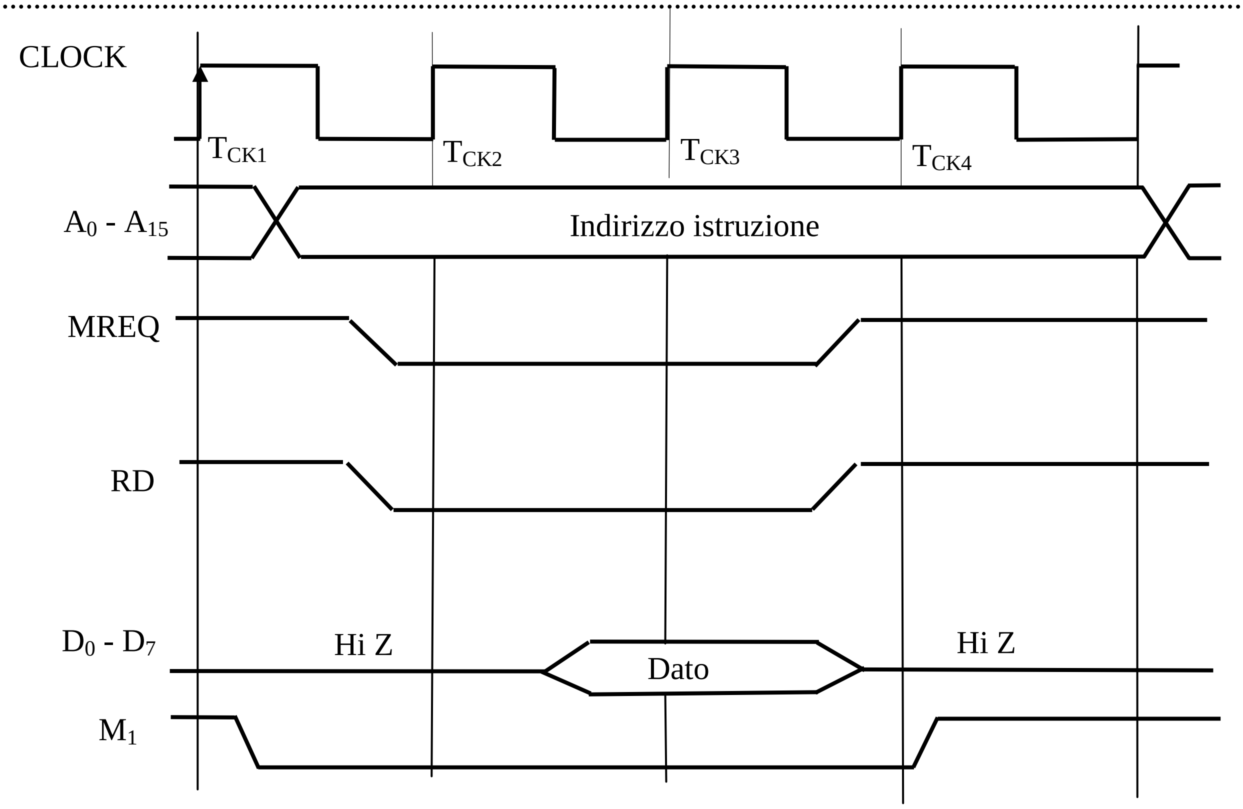


Fig. 5.8 - Ciclo di lettura in memoria

Durante la fase di ***lettura di un dato*** la CPU esegue le seguenti operazioni (vedi Fig. 5.8):

- Predispone l'indirizzo del dato da prelevare
- Attiva RD (lettura dato). Il segnale RD, direttamente collegato alla linea OE del chip di memoria, la manda bassa (abilitazione alla lettura)
- Attiva MREQ (richiesta di accesso in memoria). L'indirizzo predisposto ed il segnale MREQ, attraverso il circuito di decodifica, mandano basso CS (abilitazione del chip di memoria)
- Se e solo se il dato è costituito dal codice operativo dell'istruzione (fase di fetch), attiva il segnale M_1 per segnalare l'operazione; in tutti gli altri casi (lettura di un operando dell'istruzione, lettura di un dato) M_1 resta a livello logico 1
- Preleva il dato dal BUS
- Disattiva tutti i segnali attivati, liberando così il BUS.

5.9.1.2. Scrittura in memoria

Quanto si dirà vale ovviamente solo per la RAM.

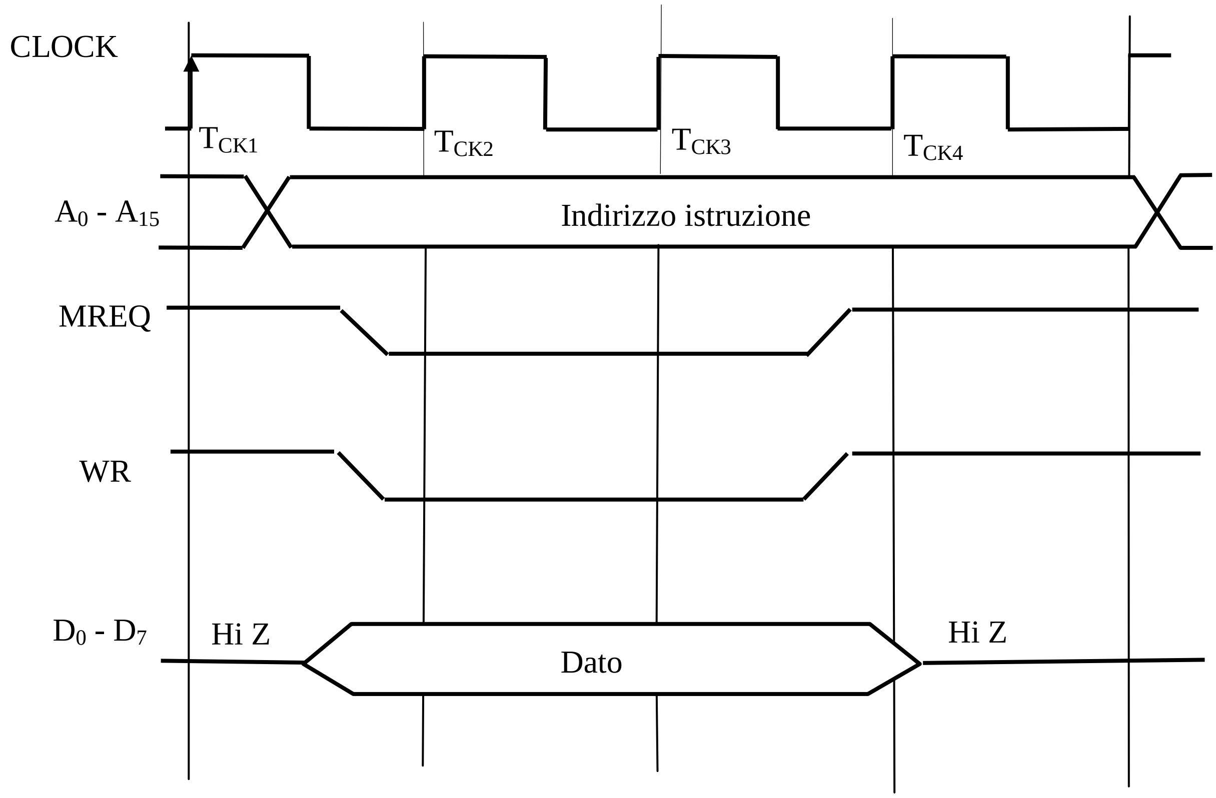


Fig. 5.9 - Ciclo di scrittura in memoria

Durante la fase di ***scrittura di un dato*** la CPU esegue le seguenti operazioni(vedi Fig. 5.9) :

- Predispone l'indirizzo del dato da memorizzare
- Predispone il dato
- Attiva WR (Scrittura dato). Il segnale WR, direttamente collegato alla linea WE del chip di memoria, la manda bassa (abilitazione alla scrittura)
- Attiva MREQ (richiesta di accesso in memoria). L'indirizzo predisposto ed il segnale MREQ, attraverso il circuito di decodifica, mandano basso CS (abilitazione del chip di memoria)
- Aspetta che la RAM immagazzini il dato
- Disattiva tutti i segnali attivati, liberando così il BUS.

Sia per la lettura che per la scrittura il BUS DATI, a parte il breve intervallo di tempo durante il quale trasporta il dato da o per la CPU, deve rimanere in uno stato di alta impedenza (Hi Z), in modo da essere sempre pronto per ogni operazione di trasferimento, anche verso altri dispositivi.

5.9.2. Temporizzazione di una porta di I/O

Le operazioni di lettura su una porta di ingresso o di scrittura su una porta di uscita hanno temporizzazioni molto simili a quelle viste per le memorie durante le stesse fasi. Vediamo le differenze.

5.9.2.1. Lettura da una porta di ingresso

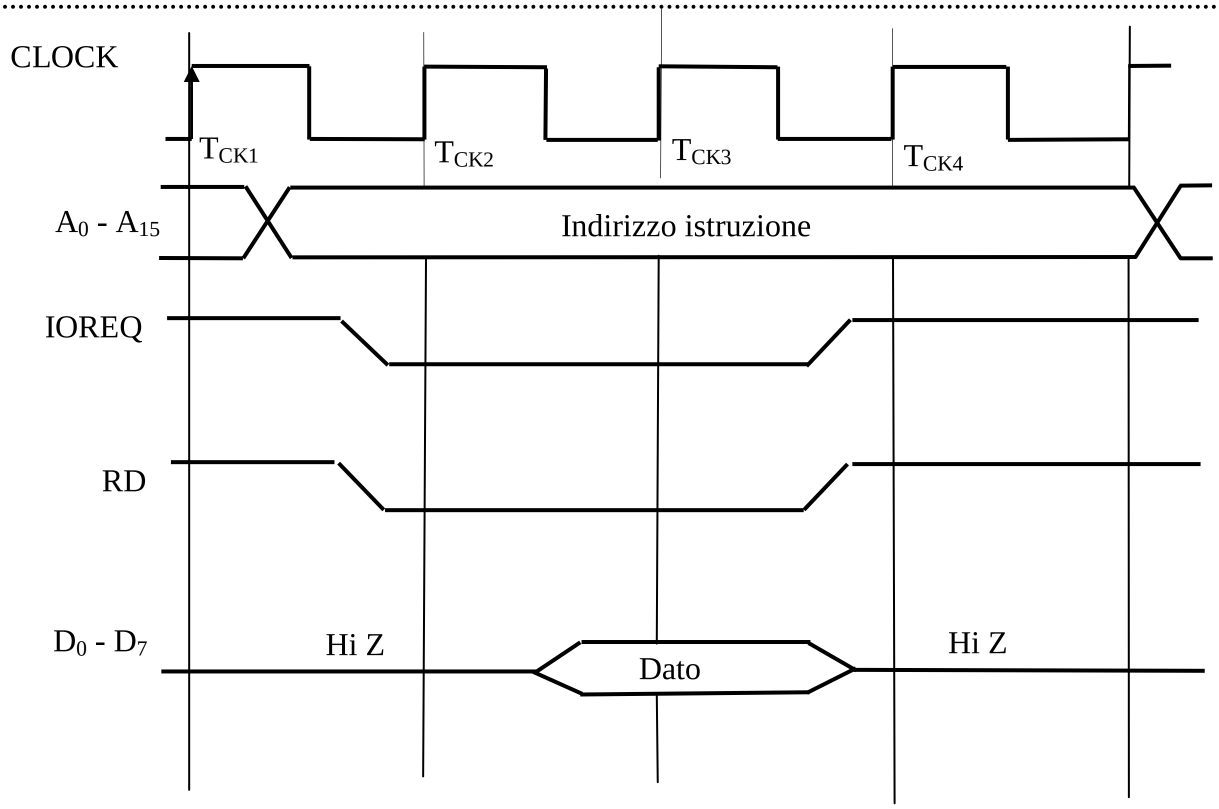


Fig. 5.10 - Ciclo di lettura su una porta di ingresso

Durante la fase di ***lettura di un dato*** la CPU esegue le seguenti operazioni (vedi Fig. 5.10) :

- Predispone l'indirizzo della porta da cui prelevare il dato
- Attiva RD (lettura dato). Quando la porta è di tipo programmabile (lettura/scrittura), il segnale RD, direttamente collegato alla linea OE della porta, la manda bassa (abilitazione alla lettura)
- Attiva IOREQ (richiesta di accesso ad una porta). L'indirizzo predisposto ed il segnale IOREQ, attraverso il circuito di decodifica, mandano basso CS (abilitazione della porta)
- Preleva il dato dal BUS
- Disattiva tutti i segnali attivati, liberando così il BUS.

5.9.2.2. Scrittura su una porta di uscita

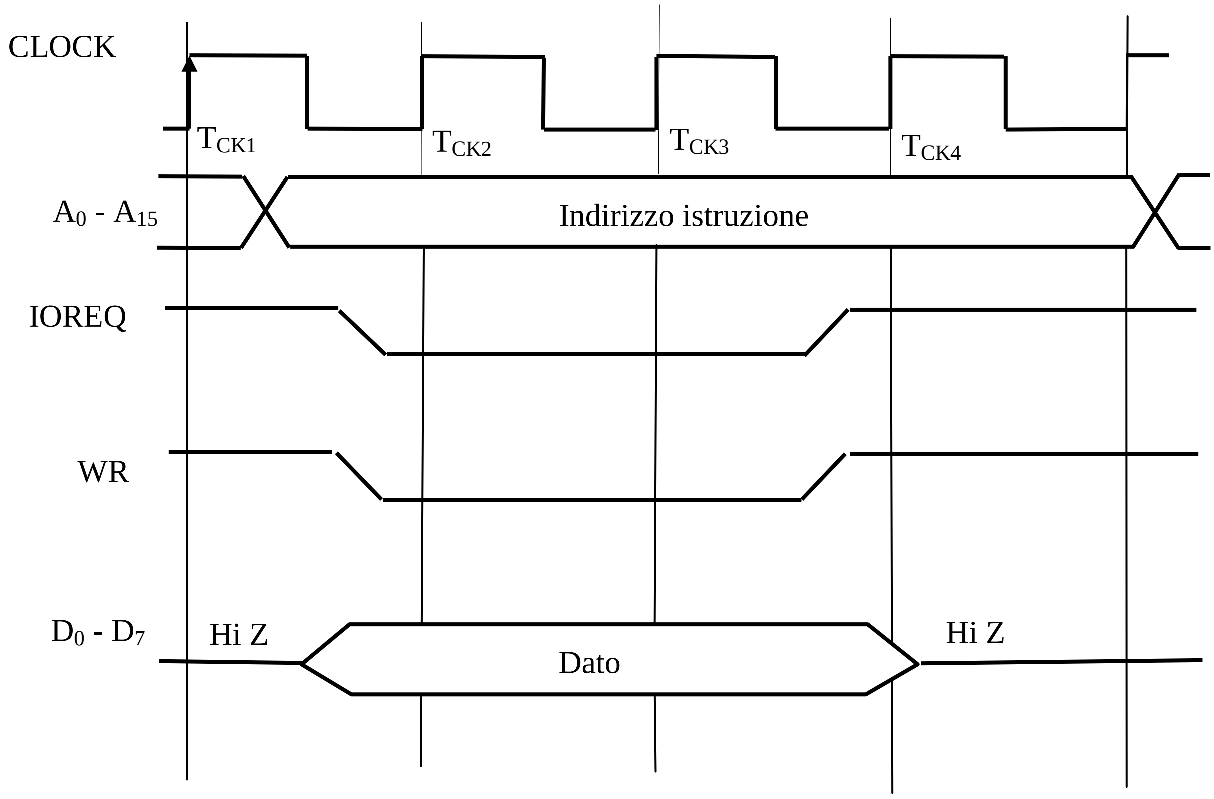


Fig. 5.11 - Ciclo di scrittura su una porta di uscita

Durante la fase di ***scrittura di un dato*** la CPU esegue le seguenti operazioni (vedi Fig. 5.11):
- Predispone l'indirizzo della porta cui inviare il dato
- Predispone il dato
- Attiva WR (Scrittura dato). Quando la porta è di tipo programmabile (lettura/scrittura), il segnale WR, direttamente collegato alla linea WE della porta, la manda bassa (abilitazione alla scrittura)
- Attiva IOREQ (richiesta di accesso ad una porta). L'indirizzo predisposto ed il segnale IOREQ, attraverso il circuito di decodifica, mandano basso CS (abilitazione della porta)
- Aspetta che la porta immagazzini il dato
- Disattiva tutti i segnali attivati, liberando così il BUS.

6. ELEMENTI DI PROGRAMMAZIONE IN ASSEMBLY

6.1. Concetti introduttivi

Abbiamo accennato che un programma (il software) è costituito da una successione di istruzioni che il μP esegue, una alla volta, in due fasi: quella di fetch e quella di execute. Ogni singola istruzione è formata da un ***codice operativo*** che rappresenta "cosa" deve fare la CPU (ovvero l'operazione che essa deve eseguire) e da un ***operando*** che può essere un dato sul quale si esegue l'operazione oppure un indirizzo di una locazione di memoria o di un periferico di I/O che costituiscono il sistema. Il progettista del software scrive il programma in base alle istruzioni che sono proprie del microprocessore impiegato (set di istruzioni) con un linguaggio, chiamato Assembly, che è di tipo mnemonico: ogni ***codice operativo*** viene identificato con un termine sintetico chiamato ***mnemonico*** che rappresenta l'operazione che deve essere eseguita. Un programma steso in ***Assembly*** è dunque costituito da una successione di righe di istruzioni scritte con una precisa sintassi; la generica linea di istruzione è strutturata nel seguente modo:

label: cod. operativo mnemonico operando ;commento

dove:

a) ***il codice operativo mnemonico*** identifica l'operazione che la CPU esegue. Esempi di codici mnemonici sono **LD** (per le operazioni di caricamento, ovvero di **L**oa**D**), **IN** (per le operazioni di **IN**put), **OUT** (per le operazioni di **OUT**put), **INC** e **DEC** (per le operazioni di **IN**-**C**remento e **DEC**remento del contenuto di un registro), ecc. Come si vede il termine "mnemonico" indica il fatto che il codice quasi sempre esprime abbastanza chiaramente, anche se sinteticamente, l'operazione da svolgere.
b) l'***operando*** può essere ***un dato*** sul quale si esegue l'operazione oppure ***un indirizzo*** di una locazione di memoria o di un periferico di I/O che costituiscono il sistema.
c) la ***label*** è una ***etichetta*** che identifica univocamente una linea di programma; tale identificazione risulta indispensabile per poter consentire le istruzioni di salto (incondizionato e condizionato) e la gestione dei sottoprogrammi chiamati anche subroutines. Essa deve terminare con il carattere "**:**".
d) il ***commento***, non essenziale per il funzionamento del programma, è molto utile perché esprime succintamente l'operazione svolta rendendo più comprensibile il listato Assembly.

Il programma così strutturato, scritto usando i codici mnemonici relativi al set di istruzione del μP impiegato, prende il nome di ***programma sorgente***; è evidente che esso, comprensibile per il programmatore, non lo è per la CPU; infatti la CPU è concepita per "capire" solo il linguaggio binario costituito da una successione di bit 0 e 1, che viene chiamato ***linguaggio macchina***. E' dunque necessario tradurre il programma sorgente in linguaggio macchina e tale operazione di traduzione viene eseguita da un programma chiamato ***Assembler*** o programma **assemblatore** che provvede a trasformare ogni istruzione scritta in mnemonico in una serie di codici operativi espressi in binario. Ciò che si ottiene dopo la "traduzione" è il ***programma oggetto***. L'assemblatore, nell'eseguire la traduzione, segnala anche eventuali errori di sintassi commessi nella stesura del sorgente. Nei prossimi paragrafi verranno illustrate ***le principali istruzioni*** relative all'Assembly dello Z 80 che ci consentiranno di scrivere semplici programmi applicativi. ***Per una panoramica completa si rinvia alla consultazione delle tabelle riassuntive delle istruzioni.***
Un programma scritto in Assembly prevede anche l'uso delle cosiddette ***direttive*** necessarie per determinare alcune operazioni da parte del programma assemblatore. Tali direttive, che vengono anche dette ***pseudo-istruzioni*** perché non generano un codice numerico, sono interpretate dall'assemblatore come dei comandi. Quelle più semplici e di uso più comune sono EQU, ORG ed END.

Concludiamo questo paragrafo precisando che nel seguito indicheremo con:
- "**r**" il generico registro ad 8 bit;
- "**rr**" la generica coppia di registri a 16 bit;
- "**n**" un dato ad 8 bit;
- "**nn**" un dato a 16 bit;
- "**cc**" una condizione logica.

Inoltre, se i dati ad 8 o 16 bit vengono espressi in esadecimale, ***per una corretta sintassi*** essi devono avere il suffisso "H"; quando la prima cifra è una delle sei lettere (A – F), bisogna anteporre alla lettera, il simbolo "0". Ad esempio se il dato da trattare fosse B7H, lo si scrive 0B7H.

6.2.Alcune considerazioni preliminari

Per capire meglio in che modo il µP gestisce le operazioni per svolgere l'attività prevista, si consideri il seguente esempio, nel quale viene ipotizzato di effettuare la somma di due numeri. L'idea è che, sapendo bene quello che deve fare il µP, quali risorse deve impiegare e come deve gestire il traffico di informazioni, sia più facile poi capire e assimilare i concetti esposti nei paragrafi seguenti.
Si faccia riferimento allora alla Fig. 6.1, nella quale viene riportata una parte della struttura interna del µP (vedi Fig. 4.1 del Cap. 4) con il suo collegamento esterno verso la memoria dati..

ROM		
IND.	**DATO**	**ISTR.**
0000	3A	a)
0001	00	
0002	F0	
0003	47	b)
0004	3A	c)
0005	01	
0006	F0	
0007	80	d)
0008	32	e)
0009	02	
000A	F0	

RAM		
IND.	**DATO**	**SIGN.**
F000	3	Add. 1
F001	4	Add. 2
F002	7	Risult.

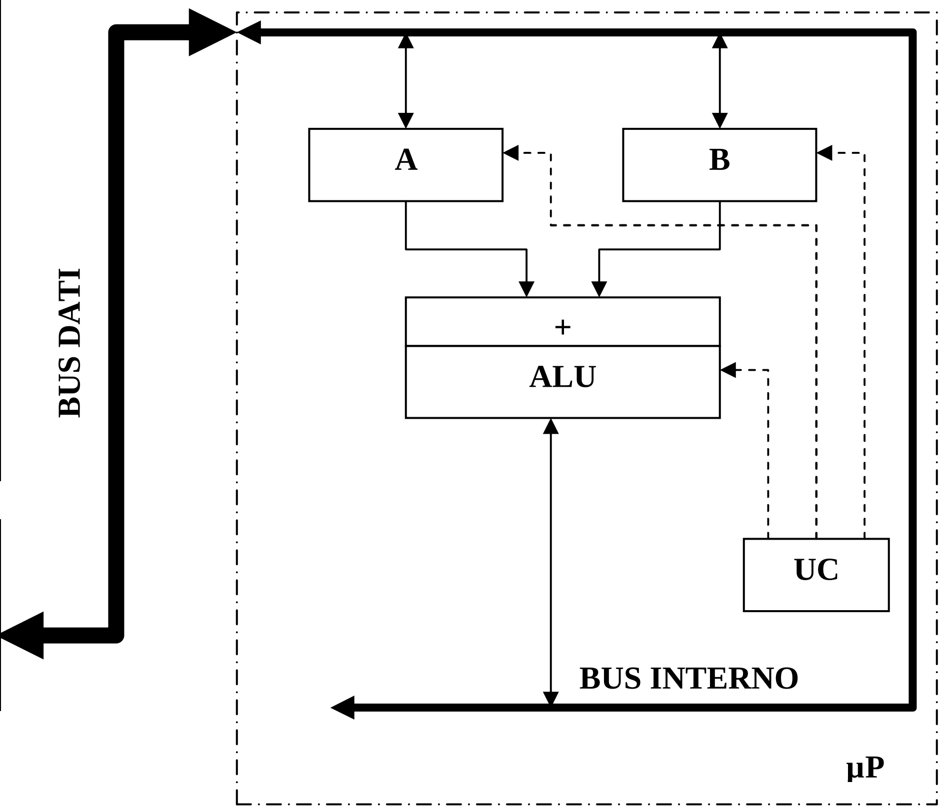


Fig. 6.1

Nella Fig. 6.1 vengono rappresentati:
1) Una memoria RAM da dove prelevare i due addendi e dove memorizzare il risultato. Ipotizzando che addendi e risultato non superino la lunghezza di un byte, occorrono tre locazioni: ad esempio quelle con indirizzo F000, F001 e F002; le prime due per gli addendi e la terza per il risultato;
2) Una Unità logico-aritmetica (ALU) che effettua la somma;
3) Due registri A e B che mantengono i due addendi durante l'operazione di somma. E' previsto che, dopo che è stata effettuata la somma, il risultato venga automaticamente memo-

rizzato nel registro A, sostituendosi all'addendo in esso contenuto che ormai non serve più.

4) Una memoria ROM che contiene le istruzioni (programma) necessarie per svolgere tutte le operazioni richieste. Supponiamo che il nostro programma sia memorizzato a partire dalla locazione 0000. Ovviamente, gli indirizzi della ROM sono separati da quelli della RAM. Questo consente al µP, che non sa distinguere fisicamente tra ROM e RAM, di affidarsi solo all'indirizzo per riconoscere la locazione (della ROM o della RAM) da utilizzare per il trasferimento del dato;
5) Una Unità di controllo (UC) che gestisce le attività, dando il consenso ai trasferimenti dei dati e, al momento opportuno, all'operazione di somma.

Perché sia possibile svolgere l'attività prevista (somma di due numeri), occorre una sequenza ordinata di istruzioni (programma) che, passo dopo passo, consenta di realizzare tutte le operazioni necessarie; questo programma risiede nella memoria ROM. Non ci si meravigli che per effettuare una semplice somma di due numeri occorrano più istruzioni. Il µP, infatti, è un operatore non intelligente che si limita ad eseguire, sotto la dettatura del programma, solo microistruzioni elementari. Il suo pregio consiste nel fatto che può eseguire milioni (o addirittura miliardi) di queste microistruzioni ogni secondo, senza stancarsi e senza sbagliare mai.
Nel nostro caso la sequenza di istruzioni che il µP deve eseguire è la seguente:

a) ***Prima istruzione*** - Prelevare dalla RAM all'indirizzo F000 il primo addendo e portarlo nel registro A (Accumulatore);
b) ***Seconda istruzione*** - Spostare questo dato dal registro A al registro B. Questo perché il dato proveniente dalla memoria arriva sempre nel registro A, per cui tale registro deve essere sempre libero per poter ricevere il nuovo dato;
c) ***Terza istruzione*** - Prelevare dalla RAM all'indirizzo F001 il secondo addendo e portarlo nel registro A. Dopo questa operazione abbiamo in B il primo addendo ed in A il secondo;
d) ***Quarta istruzione*** - Eseguire la somma tra il contenuto di A ed il contenuto di B, mettendo il risultato in A.
e) ***Quinta istruzione*** - Trasferire il dato presente in A (risultato) nella RAM all'indirizzo F002.

Nella tabella seguente vengono indicate le istruzioni, sia in assembly che in linguaggio macchina, necessarie per eseguire le cinque operazioni previste. Per adesso non è importante capire come sono state individuate tali istruzioni, visto che non conosciamo ancora, a questo punto del corso, il linguaggio assembly. Quello che conta è cercare di avere un'idea di come è organizzata, in una struttura a µP, l'attività necessaria per risolvere un determinato problema.

Operazione	**Linguaggio Assembly**	**Linguaggio macchina**		
a)	LD A,(0F000H)	3A	00	F0
b)	LD B,A	47		
c)	LD A,(0F001H)	3A	01	F0
d)	ADD A,B	80		
e)	LD (0F002H)	32	02	F0

Fermo restando che l'assembly Z80 sarà studiato nei paragrafi successivi, diamo un'occhiata alla struttura delle istruzioni, aiutandoci con la tabella precedente. Nel linguaggio assembly l'istruzione presenta sempre un codice operativo, un codice, cioè, che mnemonicamente ricorda quello che l'istruzione deve fare. Ad esempio il codice **LD** (che deriva dall'inglese **"**to **L**oa**D"**: trasferire) indica il trasferimento di un dato, da un registro all'altro o da un registro alla memoria e viceversa. Il codice **ADD** (che deriva dall'inglese "to **ADD**": sommare) indica l'operazione di somma.
Il codice operativo può essere l'unico componente dell'istruzione o può essere seguito da uno o due operandi. Gli operandi sono quantità numeriche che rappresentano i dati sui quali deve essere effettuata l'operazione prevista dal codice operativo. Quando, nelle istruzioni di trasferimento, gli operandi sono due, essi sono separati da una virgola ed il primo rappresenta la destinazione mentre il

secondo rappresenta l'origine. Una istruzione può essere preceduta da una etichetta, detta ***label***, che rappresenta un modo mnemonico per riconoscere tale istruzione, soprattutto quando la sequenza di operazioni da svolgere deve essere interrotta da un salto.
Nel linguaggio macchina, l'istruzione ha una struttura pressappoco uguale a quella vista per l'assembly. Le difficoltà nascono dal fatto che non esistono più i codici mnemonici (facili da ricordare) ma solo codici numerici, espressi per di più in esadecimale. Comunque esiste sempre un codice operativo e possono esistere uno o più operandi. Una cosa importante da sottolineare è che tra le istruzioni in assembly e quelle in linguaggio macchina c'è un rapporto 1 a 1, cioè ad una istruzione assembly corrisponde sempre una sola istruzione in linguaggio macchina.
Per quanto difficoltose da gestire, le istruzioni in linguaggio macchina sono le uniche che il µP sa interpretare, per cui l'assembly, comodo per noi, ha bisogno di una traduzione (assemblaggio) che generi i codici numerici da inserire nella memoria di programma (ROM).
A questo punto è possibile interpretare , nella Fig. 6.1, i codici numerici visibili nella ROM e i dati visibili nella RAM. Si noti che le istruzioni non hanno una lunghezza fissa, ma possono essere, per lo Z80, di 1, 2, 3 o 4 byte. Si noti ancora che i codici sono memorizzati in binario ma, per nostra comodità, vengono indicati in esadecimale.

6.3. Direttive all'assemblatore

6.3.1. Direttiva EQU (abbreviazione di **EQU**al ovvero uguale)

E' una pseudo-istruzione che consente al programmatore di ***associare ad una variabile un nome simbolico***. Tale possibilità rende il listato di più facile comprensione. Ad esempio se il valore 00H rappresenta l'indirizzo di un periferico di ingresso si può associare a tale valore un nome identificativo:

PERIN EQU 00H ;associa al valore 00H il nome simbolico PERIN

In tal modo l'anonimo valore 00H può essere indicato nel programma con l'abbreviazione PERIN che esprime sicuramente in modo più palese che si sta operando su un indirizzo di un periferico di ingresso. Analogamente se i valori numerici 9000H e A000H rappresentano due indirizzi di memoria, uno sorgente e l'altro destinazione, utilizzati per un movimento di dati, si possono associare a tali valori due nomi identificativi:

SORGENTE EQU 9000H
DESTINAZIONE EQU A000H

6.3.2. Direttiva ORG (abbreviazione di **OR**i**G**in ovvero origine)

E' una pseudo-istruzione che consente al programmatore di indicare all'assemblatore ***l'indirizzo iniziale di partenza in ROM del programma oggetto***. Per comprendere la necessità di utilizzare tale direttiva ricordiamo che, al momento dell'accensione, il Contatore di Programma punta automaticamente alla prima locazione EPROM di indirizzo 0000H per leggere la prima istruzione. Consideriamo inoltre che, senza indicazioni contrarie, il programma oggetto viene memorizzato in EPROM a partire proprio da 0000H; sembrerebbe allora naturale lasciare le cose in questo modo. In realtà bisogna considerare che le locazioni EPROM immediatamente successive all'indirizzo 0000H sono riservate per gestire le "interruzioni", tecnica di colloquio che verrà esaminata in successivamente. Pertanto, nella gestione di periferici di IN che colloquiano con il µP con la tecnica delle interruzioni, all'inizio di ogni listato Assembly devono esserci opportune indicazioni per l'Assemblatore. Tali indicazioni dovranno consentire la memorizzazione dei codici esadecimali del programma oggetto, a partire da una locazione EPROM che non interferisca con la zona riservata alle interruzioni. Siccome gli indirizzi interessati sono 0038H per interruzione mascherabile modo 1 e 0066H per interruzioni non mascherabili, normalmente si prevede di allocare l'inizio del programma oggetto a partire dall'indirizzo 0100H.

Esaminiamo in dettaglio come si procede. Supponiamo quindi che le prime due righe del programma sorgente contengano

JP 0100H
ORG 0100H

All'accensione, il Program Counter punta al codice operativo della prima istruzione all'indirizzo iniziale EPROM 0000H. La ***prima istruzione*** del programma è ***JP 0100H*** ovvero un salto incondizionato alla locazione EPROM 0100H.
Si verifica quanto segue:

- la CPU esegue la prima istruzione, di salto a 0100H;
- l'esecuzione di questa prima istruzione, carica nel PC il nuovo indirizzo 0100H; vengono quindi eseguite in sequenza tutte le istruzioni del programma che, grazie alla direttiva ORG 0100H, erano state memorizzate dall'assemblatore a partire da questa locazione.

Se, come istruzioni iniziali di un qualsiasi programma, si scrivono le due istruzioni precedenti, indipendentemente dalla lunghezza del programma ovvero indipendentemente dall'ampiezza dell'area EPROM occupata, si salta la prima parte della EPROM riservata alla gestione delle interruzioni.

6.3.3. Direttiva END

Questa pseudoistruzione viene inserita in coda al listato che costituisce il programma sorgente ed indica all'Assemblatore la fine della traduzione dal sorgente al programma oggetto.

6.4. Istruzioni di IN, OUT e LOAD

6.4.1. Generalità

Nella Fig. 6.1 sono schematicamente rappresentati i blocchi funzionali di un generico sistema a µP tra i quali è possibile lo scambio di dati gestiti direttamente dal progettista del software. Indichiamo di seguito le possibili operazioni di lettura e scrittura che la CPU può eseguire con i blocchi specificati.

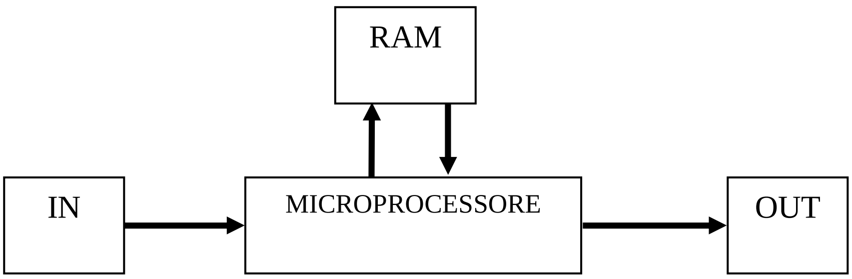


Fig. 6.1

- Se il µP vuole acquisire un dato presente sul periferico di ingresso (***lettura dal periferico di ingresso***) il programma deve prevedere l'esecuzione di una ***istruzione di INPUT***.
- Se il µP vuole inviare un dato al periferico di uscita (***scrittura sul periferico di uscita***) il programma deve prevedere l'esecuzione di una ***istruzione di OUT***.
- Se il µP vuole prelevare un dato da una locazione di memoria RAM (***lettura da locazione RAM***) il programma deve prevedere l'esecuzione di una ***istruzione di LOAD***.
- Se il µP vuole memorizzare un dato in una locazione di memoria RAM (***scrittura in locazione RAM***) il programma deve prevedere l'esecuzione di una ***istruzione di LOAD***.

Dunque, il colloquio con i periferici di I/O avviene attraverso l'esecuzione delle istruzioni di IN ed OUT mentre il colloquio con le memoria RAM avviene attraverso l'esecuzione di alcuni tipi di istruzioni di LOAD.

6.4.2. Istruzione IN e OUT

L'istruzione di INPUT consente al µP la **lettura** di un dato presente sul **periferico di ingresso**.
Trattandosi di un'istruzione di **lettura da periferico,** il µP durante l'esecuzione attiva i segnali di controllo **RD e IORQ**.

La sintassi è: ***IN destinazione, (sorgente)***
dove:
- La destinazione è l'**Accumulatore**
- La sorgente è **l'indirizzo del periferico di ingresso**

Dunque, per l'esecuzione dell'istruzione bisogna conoscere l'indirizzo del periferico. Il dato letto, ovvero il dato in ingresso al µP, giunge nel registro Accumulatore.

L'istruzione di OUTPUT consente al µP la **scrittura** di un dato sul **periferico di uscita.**
Trattandosi di un'istruzione di **scrittura su periferico,** il µP durante l'esecuzione attiva i segnali di controllo **WR e IORQ.**

La sintassi è: ***OUT (destinazione), sorgente***
dove:
- La destinazione è **l'indirizzo del periferico di uscita**
- La sorgente è l'**Accumulatore**

Dunque, per l'esecuzione dell'istruzione bisogna conoscere l'indirizzo del periferico. Il dato scritto, ovvero il dato in uscita dal µP, parte dal registro Accumulatore.

Ricordiamo quanto già detto precedentemente: per indirizzare i periferici di I/O si utilizzano le 8 linee meno significative del bus indirizzi del µP (A_0 – A_7); pertanto questi indirizzi sono espressi da dati binari ad 8 bit. ***Nelle istruzioni di IN e OUT l'indirizzo delle periferiche viene specificato con un dato ad 8 bit, espresso in esadecimale, scritto tra parentesi tonde.***

Esempio 1:
IN A, (07H) ;il dato presente sul periferico di ingresso di indirizzo 07H viene memorizzato nell'accumulatore.

Esempio 2:
OUT (00H) , A ;il dato memorizzato nell'accumulatore viene inviato sul periferico di uscita di indirizzo 00H.

Come si nota, gli indirizzi dei periferici sono stati indicati tra parentesi tonda.

Esempio 3. Acquisire un dato da un periferico di ingresso, costituito da 8 switch, di indirizzo 00H e trasferirlo su un periferico di uscita, costituito da 8 diodi led, di indirizzo 01H.
IN A, (00H)
OUT (01H), A

Esempio 3-bis. Se si volesse utilizzare la pseudo-istruzione EQU, il segmento di programma dell'Esempio 3 andrebbe scritto come segue:
SW EQU 00H
LED EQU 01H
IN A, (SW)
OUT (LED), A

In questo caso la direttiva EQU assegna ai simboli SW e LED il valore dei due indirizzi; le istruzioni di IN e OUT risultano indubbiamente più leggibili.

Le istruzioni di IN e OUT non modificano alcun flag.

6.4.3. Istruzioni di LOAD

Rispetto alle precedenti istruzioni, dove abbiamo visto che destinazione e sorgente sono univocamente determinate, per le istruzioni di LOAD la situazione è più articolata. Definiamo la sintassi specificando le possibili destinazioni e sorgenti.

La sintassi è: ***LD destinazione, sorgente***
dove:

- La destinazione può essere
  - un registro interno della CPU
  - una locazione di memoria RAM
- La sorgente può essere
  - un dato
  - il contenuto di un registro interno della CPU
  - il contenuto di una locazione di memoria RAM

6.4.3.1. Istruzioni di caricamento.

Considerando le possibili sorgenti e le diverse destinazioni si possono avere i seguenti casi di caricamento:

1) caricamento di un **dato** in un **registro interno**
2) caricamento di un **dato** in una **locazione di memoria**
3) caricamento del contenuto di un **registro** in un altro **registro**
4) caricamento del contenuto di un **registro** in una **locazione di memoria**
5) caricamento del contenuto di una **locazione di memoria** in un **registro**
6) caricamento del contenuto di una **locazione di memoria** in un'altra **locazione di memoria**

Tra le possibili combinazioni su riportate è facile riconosce che i casi 2) e 4) sono relativi ad una operazione di scrittura in memoria mentre il caso 5) è una operazione di lettura dalla memoria.
Esaminiamo separatamente questi sei casi

1) destinazione: registro interno
sorgente: dato

La sintassi è : **LD r, n** *oppure* **LD rr, nn**
In uno dei singoli registri interni ad 8 bit A, B, C, D, E, H, L si può caricare solo un dato ad 8 bit; se invece i registri si utilizzano accoppiati, nelle coppie BC, DE ed HL si possono caricare dati a 16 bit.
Esempi:
LD B, 05H ;in B si carica il valore numerico 5
LD C, 0AH ;in C si carica il valore numerico 10
LD HL, 8000H ;nella coppia HL si carica il valore numerico 8000H (80 in H e 00 in L)
Ricordiamo quanto già detto precedentemente: per indirizzare le locazioni di memoria si utilizzano le 16 linee del bus indirizzi del μP ($A_0 - A_{15}$).
L'ultima istruzione di Load vista sopra potrebbe essere quella necessaria per memorizzare il dato a 16 bit 8000H identificativo di un indirizzo di locazione RAM.

2) **destinazione: locazione di memoria**
sorgente: dato

La sintassi è : **LD (HL), n**
Il caricamento di un dato in una locazione di memoria è la tipica operazione di scrittura in RAM.
Essendo un'istruzione di **scrittura in memoria** il µP, durante la sua esecuzione, attiva i segnali di controllo **WR e MREQ.**
Per eseguire tale operazione bisogna specificare l'indirizzo della locazione RAM; in genere si procede come segue.
Si sceglie la coppia di registri HL e, con una istruzione di load come quella riportata al punto 1), si carica l'indirizzo della locazione RAM dove si vuole memorizzare il dato. Una volta che nella coppia HL è presente tale indirizzo (si dice che la coppia HL è utilizzata come puntatore di memoria) si scrive l'istruzione in esame.

Esempio: scrivere nella locazione RAM di indirizzo 8000H il dato 03H
LD HL,8000H ;in HL indirizzo della locazione RAM
LD (HL), 03H ;nella locazione puntata da HL si memorizza il valore 03H

Si nota che l'indirizzo delle locazioni di memoria viene specificato con un dato a 16 bit, espresso in esadecimale, scritto tra parentesi tonde.

3) **destinazione: registro**
sorgente: registro

La sintassi è : **LD r_1 , r_2**
Questa istruzione di LOAD consente il trasferimento di dati tra i vari registri interni.

Esempio:
LD B, A ; si trasferisce in B il contenuto memorizzato in A.

4) **destinazione: locazione di memoria**
sorgente: registro

Il caricamento del contenuto di un registro in una locazione di memoria è una tipica operazione di scrittura in memoria.
Essendo un'istruzione di **scrittura in memoria** il µP, durante la sua esecuzione, attiva i segnali di controllo **WR e MREQ.**

La sintassi é**: LD (HL), r** oppure **LD (BC), A** oppure **LD (DE), A** oppure **LD (nn), A**
Se si utilizza HL come puntatore di memoria, il registro che contiene il dato da memorizzare può essere uno qualsiasi dei registri interni. Se, invece si utilizzano come puntatori le coppie BC o DE, il registro che consente la memorizzazione del dato deve essere necessariamente A. Anche nel caso in cui l'indirizzo viene indicato esplicitamente con un valore numerico, il registro contenente il dato da memorizzare deve essere necessariamente A.

Esempio: scrivere nella locazione RAM di indirizzo 8000H il dato 03H preventivamente salvato nel registro C
LD C, 03H
LD HL, 8000H
LD (HL) ,C ;nella locazione puntata da HL si scrive il valore memorizzato nel registro C.

5) destinazione: registro
sorgente: locazione di memoria

Il caricamento del contenuto di una locazione di memoria in un registro è una tipica operazione di lettura da memoria.
Essendo un'istruzione di **lettura in memoria** il µP, durante la sua esecuzione, attiva i segnali di controllo **RD e MREQ.**

La sintassi é: **LD r, (HL)** oppure **LD A, (BC)** oppure **LD A, (DE)** oppure **LD A, (nn)**
Se si utilizza HL come puntatore di memoria, il registro che contiene il dato da memorizzare può essere uno qualsiasi dei registri interni. Se, invece si utilizzano come puntatori le coppie BC o DE, oppure l'indirizzo viene indicato esplicitamente con un valore numerico, il registro di destinazione deve essere necessariamente A.

Esempi:
LD D, (HL) ;nel registro D si trasferisce il contenuto della locazione RAM puntata da HL.
LD A, (8000H) ;nel registro A si trasferisce il contenuto della locazione RAM il cui indirizzo è 8000H

6) destinazione: locazione di memoria
sorgente: locazione di memoria

Il caricamento del contenuto di una locazione di memoria in un'altra locazione di memoria è una tipica operazione di trasferimento dati all'interno della RAM.
Supponiamo che il dato contenuto nella locazione di indirizzo 8000H debba essere spostato nella locazione 8010H. Tale trasferimento non può avvenire direttamente tra le due locazioni; è necessario trasferire il dato contenuto nell'indirizzo di partenza in uno dei registri interni del µP e poi da qui scriverlo nella locazione di indirizzo di arrivo.
Si utilizzano allora due coppie di registri interni come puntatori di memoria: per esempio la coppia HL come puntatore dell'indirizzo sorgente e la coppia DE come puntatore dell'indirizzo di destinazione; si utilizza infine uno dei rimanenti registri interni come contenitore intermedio del dato da spostare.

Esempio 1:
LD HL, 8000H ;in HL indirizzo di partenza
LD DE, 8010H ;in DE indirizzo di arrivo
LD A, (HL) ;in A il contenuto della locazione di partenza
LD (DE), A ;che viene trasferito nella locazione di arrivo

Esempio 2:
Se non si vuole usare i registri puntatori, si può procedere come segue.
LD A, (8000H) ;in A il contenuto della locazione di partenza
LD (8010H), A ;che viene trasferito nella locazione di arrivo

La differenza tra questi ultimi due esempi è costituita dal fatto che nel primo, e solo in esso, l'impiego di registri puntatori (che possono essere aggiornati) consente di ripetere più volte l'operazione utilizzando un ciclo.

6.4.3.2. Istruzioni di PUSH e POP

Può capitare, nell'ambito della stesura di un programma, la necessità di ***salvare temporaneamente*** dati presenti in alcuni registri, utilizzare liberamente questi ultimi per memorizzare altri dati e poi ***ripristinare il loro contenuto iniziale***. Tale necessità è praticamente indispensabile se il programma è strutturato con la presenza di sottoprogrammi: in tal caso spesso il sottoprogramma opera su re-

gistri i quali, durante l'esecuzione del programma, avevano memorizzato informazioni che non possono essere perse.
Per il salvataggio temporaneo ed il successivo ripristino di dati si ricorre all'uso delle istruzioni **PUSH** e **POP**.
Esse sono delle ***specifiche istruzioni di LOAD*** che consentono il ***salvataggio temporaneo (PUSH)*** del contenuto di una coppia di registri interni ed il ***successivo ricaricamento (POP)*** del contenuto originario nella stessa coppia di registri. Possono essere salvati i dati delle coppie di registri BC, DE ed HL nonché della coppia AF.

La sintassi è: ***PUSH rr*** *oppure* ***POP rr***
Avendo indicato con **rr** la coppia di registri il cui contenuto deve essere salvato.
I dati temporaneamente salvati vengono riposti in una particolare area di memoria RAM denominata ***area di stack***.
Per meglio capire il funzionamento delle istruzioni Push e Pop, diamo qualche dettaglio su ***Stack*** e ***Stack Pointer*** (SP).
Abbiamo già esaminato precedentemente il funzionamento del registro SP: in esso viene inizialmente caricato dal programmatore un indirizzo RAM che definisce la "base" dell'area di stack; tale operazione si esegue con l'istruzione:

LD SP, nn

dove "nn" è, appunto, l'indirizzo della base dell'area RAM di stack.
Una volta inizializzato lo SP, per ogni salvataggio eseguito, l'indirizzo in esso contenuto viene automaticamente decrementato e questo suo modo di funzionare fa sì che ***l'area RAM interessata si comporti in modo LIFO*** (l'ultimo dato scritto è il primo ad essere letto).
Per comprendere cosa succede nell'area di stack in seguito all'esecuzione delle istruzioni di PUSH e POP, esaminiamo le singole istruzioni che seguono e le relative illustrazioni; facciamo l'ipotesi che gli indirizzi RAM vadano da 8000H a FFFFH e scegliamo la base dell'area di stack in corrispondenza dell'ultimo indirizzo.

LD SP, 0FFFFH ;dopo l'esecuzione di tale istruzione viene definita la base dell'area di stack

FFFFH	Base area di stack dopo l'inizializzazione di SP

PUSH HL; ricordando che nelle coppie BC, DE ed HL il byte più significativo è memorizzato del primo registro mentre quello meno significativo è memorizzato nel secondo registro, dopo l'esecuzione di tale istruzione, ovvero dopo il salvataggio del contenuto di HL, l'area di stack si presenta come segue.

FFFDH	Byte contenuto in L
FFFEH	Byte contenuto in H
FFFFH	Base area di stack dopo l'inizializzazione di SP

PUSH DE; dopo l'esecuzione di tale istruzione, l'area di stack si presenta come segue.

FFFBH	Byte contenuto in E
FFFCH	Byte contenuto in D
FFFDH	Byte contenuto in L
FFFEH	Byte contenuto in H
FFFFH	Base area di stack dopo l'inizializzazione di SP

Dunque, al termine del salvataggio, nell'area di stack si trovano "sovrapposti", a partire dal primo, i quatto byte salvati. Altre istruzioni di PUSH continuerebbero a salvare in locazioni via via decrescenti i successivi byte.
Esaminiamo ora il caso di recupero dei dati salvati nello stack. E' evidente che non è possibile ripristinare l'originale contenuto di HL se non si provvede prima a recuperare il contenuto di DE: ***dunque l'ultimo dato salvato deve essere il primo ad essere recuperato.***

POP DE ; dopo l'esecuzione di tale istruzione in DE si ripristina il valore temporaneamente salvato e l'area di stack si presenta come segue.

FFFDH	Byte contenuto in L
FFFEH	Byte contenuto in H
FFFFH	Base area di stack dopo l'inizializzazione di SP

POP HL ; dopo l'esecuzione di tale istruzione anche in HL si ripristina il valore temporaneamente salvato e l'area di stack si presenta come segue.

FFFFH	Base area di stack dopo l'inizializzazione di SP

E' importantissimo ricordare che per un corretto salvataggio temporaneo e successivo recupero dei dati memorizzati nei registri coinvolti il numero delle istruzioni POP deve essere sempre uguale a quello delle istruzioni PUSH; inoltre l'ordine di scrittura delle istruzioni POP deve essere sempre inverso a quelle delle istruzioni PUSH.

Tutte le istruzioni di trasferimento su riportate non modificano alcun flag.

6.5. Le istruzioni aritmetiche e logiche

6.5.1. Istruzioni di Incremento e Decremento

Consentono di incrementare o decrementare ***di una unità*** il contenuto di un registro (ad 8 bit) o di una coppia di registri (a 16bit).

La sintassi è: ***INC r*** *oppure* ***INC rr***
DEC r *oppure* ***DEC rr***

Solo per la coppia HL è possibile utilizzare l'istruzione ***INC (HL)*** che consente di incrementare di una unità il contenuto della locazione di memoria puntata da HL e ***DEC (HL)*** che esegue l'operazione opposta.
Le istruzioni INC r ; INC (HL) ; DEC r modificano i flags S, Z; le istruzioni INC rr e DEC rr non modificano alcun flag.

Esempi:

LD B, 03H	;in B il valore 03H
INC B	;dopo tale istruzione in B sarà memorizzato il valore 04H
LD B, 03H	;in B il valore 03H
DEC B	;dopo tale istruzione in B sarà memorizzato il valore 02H
LD DE, 8000H	;in DE primo indirizzo RAM
INC DE	;dopo tale istruzione in DE si trova il valore 8001H
LD HL, 8000H	;in HL indirizzo RAM
INC (HL)	;supponendo che nella locazione 8000H ci sia il dato 03H, dopo tale istruzione il dato nella stessa locazione assume il valore 04H.

Quando in un ***registro ad 8 bit*** si carica un certo valore e poi si esegue un ciclo continuo di decremento, ***l'azzeramento del contenuto del registro viene segnalato dal flag di Zero.***
Quando in una ***coppia di registri a 16 bit*** si carica un certo valore e poi si esegue un ciclo continuo di decremento, ***l'azzeramento del contenuto della coppia non viene segnalato da alcun flag.*** In questo secondo caso la segnalazione dell'azzeramento deve essere prodotta con un artificio software illustrato nell'esempio di impiego della istruzione logica OR (vedi esempio n° 2 del par.6.5.5.).

6.5.2. Istruzione di confronto

L'operazione di confronto è una particolare sottrazione. Se N_1 ed N_2 sono due generici dati numerici, confrontarli tra loro significa eseguire l'operazione $N_1 – N_2$ e valutarne il risultato:
se $N_1 > N_2$ il confronto fornisce un risultato >0
se $N_1 = N_2$ il confronto fornisce un risultato =0
se $N_1 < N_2$ il confronto fornisce un risultato <0
Il risultato > = o < di zero ci fornisce l'indicazione sull'esito del confronto.
In logica cablata tale operazione viene eseguita dal circuito comparatore; se per esempio ci si riferisce a dati numerici a 4 bit, il C.I. è il 7485; per dati numerici di estensione maggiore si collegano 2 C.I. 7485 in cascata.
In logica programmata l'operazione di confronto viene eseguita dalla istruzione CP la cui sintassi è:

CP n *oppure* ***CP r*** *oppure* ***CP (HL)***

Esse consentono il confronto tra quanto memorizzato nell'accumulatore ed un dato che può essere esplicito, il contenuto in un registro o il contenuto della locazione di memoria puntata da HL. In pratica, come già accennato, è come se si effettuasse una operazione di sottrazione il cui esito non modifica il contenuto dell'accumulatore ma agisce soltanto sui flags di Zero e di Carry.
Se indichiamo con "X" il dato o il contenuto di un registro rispetto al quale si vuole eseguire un confronto, l'istruzione effettua l'operazione A – X ottenendosi, come esito del confronto, l'attivazione dei flags Z e C come riportato nella seguente tabella:

Esito del confronto	Flag di Zero	Flag di Carry
A > X	***Z = 0***	***C = 0***
A = X	***Z = 1***	***C = 0***
A < X	***Z = 0***	***C = 1***

- se A > X il risultato del confronto è maggiore di zero e ciò ***non viene segnalato*** con l'attivazione dei flags Z e C;
- se A = X il risultato del confronto è nullo e ciò ***viene segnalato con l'attivazione del flag di Zero;***
- se A < X il risultato del confronto è minore di zero e ciò ***viene segnalato con l'attivazione del flag di Carry.***

6.5.3. Istruzioni aritmetiche di uso generale.

La prima operazione che si studia sulle variabili binarie è la negazione ed essa, in logica cablata, è implementata dalla porta NOT.
In logica programmata esiste la possibilità di complementare l'intero contenuto dell'accumulatore utilizzando l'istruzione ***CPL***. Dopo l'esecuzione di questa istruzione gli 8 bit dell'accumulatore vengono negati uno ad uno ovvero si effettua il ***complemento ad uno*** del dato memorizzato nell'accumulatore.
Questa istruzione non modifica i flags S, Z e C.
Esiste anche l'istruzione ***NEG*** che, come CPL, opera sull'intero contenuto dell'accumulatore eseguendo il complemento a due del dato in esso memorizzato.
Questa istruzione modifica i flags S, Z e C.

6.5.4. Istruzioni di somma e sottrazione.

In logica cablata le operazioni aritmetiche di somma e sottrazione tra due numeri ad "n" bit vengono entrambe eseguite dal circuito sommatore; se ad esempio ci riferiamo a dati numerici a 4 bit, il C.I. è il 7483; per numeri di estensione maggiore si collegano 2 C.I. 7483 in cascata
Ricordiamo che è anche possibile eseguire sottrazioni utilizzando il circuito sommatore in quanto tale operazione viene condotta con il metodo del complemento a due, riconducendo l'operazione di differenza in somma:

$$N_1 - N_2 = N_1 + (/N_2 + 1)$$

In logica programmata queste due operazioni vengono svolte utilizzando le istruzioni ADD e SUB la cui sintassi è:

ADD A, n *oppure* ***ADD A, r*** *oppure* ***ADD A, (HL)*** *per la somma*
SUB n *oppure* ***SUB r*** *oppure* ***SUB (HL)*** *per la sottrazione.*

Le istruzioni su riportate modificano i flags S, Z e C.

L'istruzione ***ADD*** esegue la somma tra il byte contenuto in A e l'operando, che può essere un dato immediato ad 8 bit, il contenuto di un registro o il contenuto della locazione di memoria puntata da HL. Il risultato viene salvato nell'accumulatore.

Esempio: assegnati due numeri N_1 ed N_2, eseguirne la somma.
LD A, N_1 ;in A è salvato il valore N_1
LD B, N_2 ;in B è salvato il valore N_2
ADD A,B ;in A è salvato il valore del risultato N_1+N_2
Siccome il risultato viene salvato nell'accumulatore che è un registro ad 8 bit, esso non può essere superiore ad FFH, ovvero a 255; nel caso in cui il valore del risultato fosse superiore, tale circostanza viene segnalata dalla CPU con il settaggio del flag di C.

L'istruzione ***SUB*** esegue la differenza tra il byte contenuto in A e l'operando, che, come per la somma, può essere un dato immediato ad 8 bit, il contenuto di un registro o il contenuto della locazione di memoria puntata da HL. Il risultato viene salvato nell'accumulatore.
La CPU esegue la sottrazione con il metodo del complemento a due: se la differenza risulta positiva, il contenuto di A esprime il valore della operazione ed il flag di Carry resta a 0 segnalando assenza di prestito; se invece la differenza risulta negativa il contenuto di A esprime il risultato in complemento a due ed il flag di Carry si setta ad 1 segnalando un prestito.

6.5.5. Istruzioni di AND, OR e XOR

Sono istruzioni che eseguono l'operazione logica AND, OR o XOR, bit per bit, tra il contenuto dell'accumulatore ed un dato che può essere esplicito o contenuto in un registro o nella locazione di memoria puntata da HL.
Il risultato dell'operazione viene memorizzato nell'accumulatore.

La sintassi è: ***AND n*** *oppure* ***AND r*** *oppure* ***AND (HL)***

OR n *oppure* ***OR r*** *oppure* ***OR (HL)***

XOR n *oppure* ***XOR r*** *oppure* ***XOR (HL)***

Queste istruzioni modificano i flags di S e Z.

Esempio n° 1

Partendo dal contenuto iniziale salvato nell'accumulatore, isolare il bit b_2.

Contenuto iniziale dell'Accumulatore							
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	**0**	**0**	**1**	**1**	**1**	**0**	**1**

Codice **04H** utilizzato come maschera							
0	**0**	**0**	**0**	**0**	**1**	**0**	**0**

Contenuto dell'accumulatore dopo l'esecuzione dell'istruzione **AND 04H**							
0	**0**	**0**	**0**	**0**	**1**	**0**	**0**

L'esempio riportato potrebbe riferirsi ad una acquisizione di un byte da un periferico di ingresso, dove ogni singolo bit rappresenta l'uscita di un sensore. Se si ha la necessità di testare il solo stato del sensore relativo alla linea b_2, visualizzandolo su un diodo led, è chiaro che tale linea deve essere isolata. Per fare questo si esegue l'AND con il codice 04H che consente la selezione desiderata. L'istruzione "AND 04H", infatti, azzererà tutti i bit del dato contenuto nell'accumulatore tranne b_2 che conserverà il suo valore originario (0 oppure 1). Si noti che il codice-maschera coincide con il peso del bit da isolare. Siccome dopo l'operazione AND il contenuto dell'accumulatore viene perso, se si vuole evitare ciò si possono utilizzare le istruzioni PUSH e POP.
Un segmento di programma che riassume, oltre all'istruzione in esame, anche quelle studiate in precedenza, potrebbe essere il seguente:

PERIN EQU 00H ;identifica l'indirizzo 00H del periferico di ingresso (switch) con l'etichetta *"PERIN"*
PEROUT EQU 001H ;identifica l'indirizzo 01H del periferico di uscita (diodo led) con la e-

tichetta PEROUT
LD SP, nn ;inizializza la base dello stack
IN A, (PERIN) ;legge il dato presente sul periferico di ingresso e lo memorizza in A
PUSH AF ;salva il contenuto della coppia AF
AND 04H ;isola il bit b_2
OUT (PEROUT), A ;invia sul periferico di uscita il valore di b_2
POP AF ;ripristina i valori originari nella coppia AF

In definitiva con l'istruzione AND è possibile isolare uno o più bit di un byte contenuto nell'accumulatore utilizzando un opportuno codice impiegato come "maschera"; se i bit da isolare sono più di uno, il codice-maschera coincide con la somma dei pesi dei bit da isolare.

Esempio n° 2
Partendo dal contenuto iniziale della coppia di registri HL, ottenere la segnalazione di azzeramento attraverso l'attivazione del flag di Zero.
Questo problema si incontra quando, in modalità reale, si devono ottenere dei tempi di ritardo sufficientemente lunghi e per tale motivo si utilizza una coppia di registri. Per lo studio del problema si rinvia al paragrafo 6.7. relativo ai cicli di ritardo; in questo punto vogliamo solo mostrare un possibile uso dell'istruzione OR.
Nel decrementare un numero contenuto nella coppia HL si azzera prima il valore di H (byte più significativo) e poi quello di L (byte meno significativo). L'artificio consiste nel caricare nell'accumulatore il contenuto di H e di eseguire successivamente una operazione logica OR con il contenuto di L. Nell'ipotesi che in H il valore si sia annullato, anche in A il valore è nullo. Partendo da tale condizione fino a quando nel registro L il valore non è nullo, l'OR con A darà sempre risultato diverso da zero. Solo quando anche il registro L sarà azzerato, l'istruzione "OR L" azzera l'accumulatore e tale evento setta il flag di Zero che, dunque, indirettamente segnala l'azzeramento della coppia HL. Ricordiamo che l'operazione OR dà risultato nullo se e solo se i due operandi sono nulli.

DEC HL
LD A, H
OR L

Esempio n° 3
Partendo da due acquisizioni successive da una stessa porta, identificare i bit che hanno cambiato stato.
In questo caso il segmento di programma potrebbe essere il seguente:

IN A, (PERIN) ;in A primo dato acquisito
LD B, A ;che viene salvato in B
IN A, (PERIN) ;in A secondo dato acquisito
XOR B ;identifica i bit che hanno cambiato stato e li salva in A

Contenuto iniziale dell'Accumulatore Salvato in B							
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	**1**	**1**	**0**	**1**	**0**	**1**	**1**

Contenuto dell'Accumulatore Dopo il secondo IN							
1	**0**	**1**	**1**	**0**	**0**	**1**	**1**

Contenuto dell'accumulatore dopo l'esecuzione dell'istruzione **XOR B**							
0	**1**	**0**	**1**	**1**	**0**	**0**	**0**

Ricordiamo che l'XOR tra due variabili fornisce una uscita bassa se le due variabili sono uguali o una uscita alta se esse sono diverse.
I bit alti b_6, b_4 e b_3 indicano quelli che tra la prima e seconda acquisizione si sono modificati.

6.6. Le istruzioni di salto

Strutturare un programma senza la possibilità di poter prevedere dei "salti" è impensabile. Tali salti possono essere:

Salti incondizionati, la cui sintassi è : ***JP LABEL***

Salti condizionati, la cui sintassi è : ***JP cc, LABEL***

Entrambe le istruzioni non modificano i flags.

6.6.1. Salti incondizionati.

Sono quelli che vengono effettuati senza aspettare il verificarsi di alcuna condizione logica; vengono effettuati e basta. L'istruzione di destinazione del salto deve essere necessariamente identificata con una LABEL per consentire l'individuazione del punto in cui saltare.
Supponiamo di avere un programma costituito da una serie di istruzioni che eseguono una generica operazione e supponiamo che tale operazione debba essere ripetuta un numero indefinito di volte nel tempo. Se, dopo l'ultima istruzione non si prevede un salto che riconduca alla prima istruzione, la generica operazione verrà eseguita una sola volta.

Prima istruzione………………..
Seconda istruzione……………..
.
.
.
Ultima istruzione………………. *(fine programma)*

Per ripetere in modo continuo l'insieme delle istruzioni, ovvero per creare un ciclo, è indispensabile la presenza di una istruzione di salto: tale salto deve essere eseguito senza che si verifichi alcuna condizione se non quella che si è giunti al termine del programma e lo si vuole ripetere.
La sintassi dei salti incondizionati è :

JP LABEL

Ciclo: ***Prima istruzione……………………..***
Seconda istruzione……………………
.
.0
.
Ultima istruzione…………………….
JP Ciclo ***;****(il programma riprende dalla prima istruzione che è stata identificata dalla label "Ciclo")*

Riprendiamo in esame il seguente esempio, già analizzato nel paragrafo relativo alle istruzioni IN ed OUT: acquisire un dato da un periferico di ingresso, costituito da 8 switch, di indirizzo 00H e trasferirlo su un periferico di uscita, costituito da 8 diodi led, di indirizzo 01H.
Il segmento di programma è:

IN A, (00H)
OUT (01H), A

Con tale segmento si può avere l'acquisizione di ***un solo dato*** dal periferico di ingresso. Se volessimo continuamente leggere l'ingresso e trasferire in uscita i dati letti, dovremmo prevedere, dopo l'ultima istruzione, un salto incondizionato alla prima istruzione, assegnando ad essa per esempio l'etichetta LOOP:

```
LOOP:   IN A, (SW)      ;legge dato in ingresso
        OUT (LED), A    ;lo scrive in uscita
        JP LOOP         ;ripete ciclo
```

Con questo segmento di programma possiamo acquisire e pubblicare in continuazione un numero indefinito di dati. Si noti che nell'istruzione di salto "JP LOOP" l'etichetta va indicata senza i due punti.

6.6.2. Salti condizionati.

Sono quelli che vengono eseguiti solo in corrispondenza del verificarsi di una precisa condizione. La condizione è fornita dallo stato dei flags.
Il programmatore può decidere se condizionare il salto in relazione al verificarsi della "***condizione***" o della "***non condizione***":
La tabella che segue riporta alcune condizioni e "non condizioni" che possono determinare i salti condizionati relativamente ai flags che abbiamo esaminato.

ZERO (Z)	Z = 1	JP Z, label
NON ZERO (NZ)	Z = 0	JP NZ, label
RIPORTO (C)	C = 1	JP C, label
NON RIPORTO (NC)	C = 0	JP NC, label
NEGATIVO (M)	S = 1	JP M, label
POSITIVO (P)	S = 0	JP P, label

Le istruzioni di salto condizionato o incondizionato, pur utilizzando i flag, non ne modificano alcuno.

Esempio n° 1

```
        LD A, 03H       ;in A valore iniziale 3
LOOP:   INC A           ;incrementa A
        CP 6            ;e lo confronta con 6 (A – 6)
        JP C, LOOP      ;se C=1 (A<6), salta a LOOP, altrimenti prosegue uscendo dal
    ciclo.
```

La condizione (C=1) diventa falsa quando A = 6; in tal caso viene interrotta la ripetizione delle operazioni.

Esempio n° 2

	LD A, 06H	;in A valore iniziale 6
LOOP:	***DEC A***	;decrementa A
	JP NZ, LOOP	;se Z=0 (A≠0), salta a LOOP.

La condizione (Z=0) diventa falsa quando A=0; in tal caso viene interrotta la ripetizione delle operazioni. Si tenga presente che quando Z=0 ➔ A≠0 e quando Z=1 ➔ A=0; in altre parole il flag Z segnala con un 1 il verificarsi di un risultato nullo nell'ultima istruzione eseguita mentre con uno 0 segnala il verificarsi di un risultato non nullo.

6.7. Cicli di ritardo

Questi cicli sono molto impiegati come segmenti di programma che generano ritardi temporali via software; essi rappresentano un tipico esempio di struttura basata essenzialmente sull'uso dei salti condizionati.
Si procede come segue: inizialmente si carica in un registro un certo valore, il quale viene poi decrementato; l'azzeramento del contenuto del registro viene segnalato dal flag di zero. La durata del ritardo dipende, a parità di frequenza del µP, dalla grandezza del numero inizialmente caricato nel registro scelto.
Svilupperemo tre tipologie di ciclo che consentono di ottenere ritardi via via crescenti.

Esempio n° 1

Quello che segue è il caso più semplice, che genera un ritardo impiegando un singolo registro ad 8 bit.

	LD B, 07H	;in B il valore 7
LOOP:	***DEC B***	;decrementa il contenuto
	JP NZ, LOOP	;salta a LOOP se il flag di zero è ancora alto (B≠0)

Volendo ottenere con un singolo registro ad 8 bit il massimo tempo di ritardo, si deve caricare nello stesso il massimo valore numerico FFH.

	LD B, 0FFH	;in B il valore FFH
LOOP:	***DEC B***	;decrementa il contenuto
	JP NZ, LOOP	;salta a loop se il flag di zero è ancora alto (B≠0)

Proviamo a calcolare quale ritardo si riesce ad ottenere. Per fare ciò dobbiamo conoscere la frequenza di funzionamento della CPU ed il numero totale dei cicli elementari che il segmento richiede. Se, come nel caso del nostro sistema di sviluppo, la frequenza è pari a 2,5 MHz, la durata del ciclo elementare è T_{ck} = 0,4 µs.
Analizziamo ora il numero dei cicli consultando i fogli delle istruzioni:

- per l'esecuzione della prima istruzione sono richiesti 7 cicli T_{CK}
- per l'esecuzione della seconda istruzione sono richiesti 4 cicli T_{CK}
- per l'esecuzione della terza istruzione sono richiesti 10 cicli T_{CK}

Per ogni istruzione occorre calcolare quante volte viene ripetuta, in modo da poter calcolare il tempo che la CPU spende per essa. Sommando poi i tempi ottenuti per ogni istruzione, si ottiene il tempo totale che la CPU spende per eseguire l'intero segmento di programma. La tabella seguente dà un'idea di come procedere per il calcolo.

Istruzione	Num. T_{ck}	Num. Ripetizioni	Num. Tot. T_{ck}	Tempo
LD B,0FFH	7	1	7*1 = 7	7*0,4 = 2,8 µs
DEC B	4	255 (FFH)	4*255 = 1020	1020*0,4 = 408 µs
JP NZ,LOOP	10	255 (FFH)	10*255 = 2550	2550*0,4 = 1020 µs
			Tempo totale:	**1430,8 µs ≈ 1,43 ms**

Questo conteggio evidenzia che con un unico registro ad 8 bit il massimo ritardo ottenibile in modalità reale è poco più di un millisecondo. Per tale motivo verranno analizzati cicli più complessi che consentono di ottenere maggiori valori di tempi di ritardo. Tali cicli utilizzeranno più di un registro e in questo caso il conteggio del numero totale di cicli risulta laborioso; pertanto ricorreremo ad una espressione semplificata per il calcolo del ritardo che consente una rapidissima valutazione dei tempi con una ottima approssimazione.

L'espressione è: $\mathbf{T_R = 14\ n_1\ n_2 \ldots n_n\ 0{,}4}$ ***µs***

dove $\mathbf{n_1\ n_2 \ldots n_n}$ sono i valori numerici salvati negli n registri.

Esempio n° 2

Come secondo esempio generiamo un ritardo utilizzando due registri ad 8 bit, per esempio B e C; il ciclo di decremento del numero contenuto in B, ovvero 255, viene ripetuto un numero di volte pari a quello contenuto in C, ovvero 37 volte.

```
        LD C, 37H
LOOP1:  LD B, 0FFH
LOOP:   DEC B
        JP NZ, LOOP
        DEC C
        JP NZ, LOOP1
```

Applicando la formula indicata precedentemente, si ottiene:

$$Ritardo = 14\ n_1\ n_2\ 0{,}4 = 14 \times 255 \times 37 \times 0{,}4 = 52{,}8\ ms$$

Volendo calcolare il massimo tempo di ritardo ottenibile, si deve ipotizzare che anche in C sia caricato il valore FFH; in questa ipotesi si ha:

$$Ritardo = 14\ n_1\ n_2\ 0{,}4 = 14 \times 255 \times 255 \times 0{,}4 = 0{,}364\ s$$

Anche in questo caso si nota che il ritardo non va oltre i decimi di secondo.
Se vogliamo iniziare ad ottenere tempi dell'ordine dei secondi possiamo utilizzare 3 registri ad 8 bit o ricorrere ad un registro ad 8 bit ed un registro a 16 bit.
Nel caso di tre registri ad 8 bit, caricati con il numero massimo FFH, è facile verificare che si ottiene un ritardo di circa 93 sec.

Esempio n° 3

Generiamo un ritardo utilizzando un unico registro a 16 bit. A fianco alle singole istruzioni abbiamo riportato il numero dei cicli T_{ck} necessari per l'esecuzione delle stesse.

```
        PUSH AF          11
        LD DE, 0FFFFH    10
LOOP:   DEC DE           6
        LD A, D          4
        OR E             4
        JP NZ, LOOP      10
        POP AF           10
```

Rispetto al listato relativo ad un unico registro ad 8 bit si notano due istruzioni diverse:
LD A, D
OR E
Ricordiamo che il significato di queste istruzioni aggiuntive, artificio necessario in quanto l'azzeramento di un registro a 16 bit non viene segnalato da nessun flag, è stato in precedenza esaminato quando si è parlato dell'istruzione logica OR. Come conseguenza dell'artificio che coinvolge l'accumulatore, siamo costretti ad aggiungere anche le istruzioni PUSH AF e POP AF.
La tabella seguente riporta il calcolo necessario per conoscere il tempo totale impiegato dalla CPU per eseguire il segmento di programma.

Istruzione	**Num. T_{ck}**	**Num. Ripetizioni**	**Num. Tot. T_{ck}**	**Tempo**
PUSH AF	11	1	11*1 = 11	11*0,4 = 4,4 μs
LD DE,0FFFFH	10	1	10*1 = 10	10*0,4 = 4 μs
DEC DE	6	65535 (FFFFH)	6*65535 = 393210	393210*0,4 = 157284 μs
LD A,D	4	65535 (FFFFH)	4*65535 = 262140	262140*0,4 = 104856 μs
OR E	4	65535 (FFFFH)	4*65535 = 262140	262140*0,4 = 104856 μs
JP NZ,LOOP	10	65535 (FFFFH)	10*65535 = 655350	655350*0,4 = 262140 μs
POP AF	10	1	10*1 = 10	10*0,4 = 4 μs
			Tempo totale:	**629148,4 μs ≈ 0,63 s**

<u>Esempio n° 4</u>
Generiamo un ritardo utilizzando due registri di cui uno ad 8 bit ed uno a 16 bit.

```
          PUSH AF
          LD B, 02H
LOOP:     LD DE, 0FFFFH
LOOP1:    DEC DE
          LD A, D
          OR E
          JP NZ, LOOP1
          DEC B
          JP NZ, LOOP
          POP AF
```

Il programma che stiamo esaminando concettualmente prevede la ripetizione di quello sviluppato nell'esempio precedente per 2 volte, ovvero per il numero caricato nel registro B. Approssimativamente si può pertanto valutare il tempo della routine come: Ritardo = n x 0,63 = 2 x 0,63 = 1.26 s.
Il massimo tempo di ritardo realizzabile con due registri di cui uno ad 8 bit ed uno a 16 bit vale: 255 x 0,63 = 160,65 s, ovvero poco più di due minuti e mezzo.

6.8. Istruzioni orientate al bit

Quando si vuole conoscere il livello logico di uno degli 8 bit memorizzati in un registro o in una locazione di memoria puntata da HL si utilizzano le istruzioni:

<u>***BIT x, r*** *oppure* ***BIT x, (HL)***</u>

dove "x" è il numero del bit da testare; precisiamo che il numero 7 identifica il bit b_7 che è quello più significativo mentre il numero 0 identifica il bit b_0 che è quello meno significativo.
In seguito a tali istruzioni ***si verifica il livello del bit testando il flag di zero***:
se Z = 1 il bit in questione è 0, ovvero basso;
se Z = 0 il bit in questione è 1, ovvero alto.
In pratica, l'informazione ottenuta sul livello del bit testato non è subito evidente ma viene utilizzata

implicitamente per effettuare una scelta mediante un salto condizionato: tale scelta sarà diversa a seconda che il valore del bit testato sia 0 o 1.

Esempi:

BIT 3, C ; Si testa il bit b_3 del registro C
JP Z, AAA ; se tale bit è 0 si salta all'istruzione con etichetta AAA. Se il bit è 1, invece, si evita il salto e si prosegue in sequenza con l'istruzione successiva.
BIT 7, (HL) ; Si testa il bit b_7 della locazione di memoria puntata da HL
JP NZ, BBB ; se tale bit è diverso da 0 (vale a dire 1) si salta all'istruzione con etichetta BBB. Se il bit è 0, invece, si evita il salto e si prosegue in sequenza con l'istruzione successiva.

Quando si vuole forzare a 0 o ad 1 uno degli 8 bit memorizzati in un registro o in una locazione di memoria puntata da HL si utilizzano le istruzioni:

SET x, r ; forza a 1 (setta) il bit di ordine x del registro r
SET x, (HL) ; forza a 1 (setta) il bit di ordine x della locazione di memoria puntata da HL

RES x, r ; forza a 0 (resetta) il bit di ordine x del registro r
RES x, (HL) ; forza a 0 (resetta) il bit di ordine x della locazione di memoria puntata da HL

Queste istruzioni non modificano alcun flags.

6.9. Le istruzioni di rotazione e di scorrimento

6.9.1. Istruzioni di rotazione

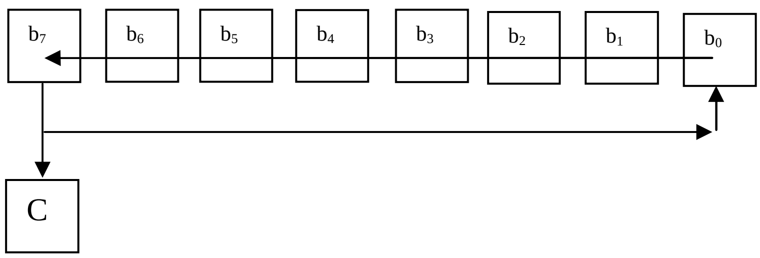
Tra le varie istruzioni di rotazione possibili esaminiamo le seguenti:

RLC (Rotate Left Circular) ovvero ***rotazione circolare verso sinistra*** di un dato contenuto in un registro o in una locazione di memoria, la cui sintassi è:

RLC r *oppure* ***RLC (HL)***

Per ogni istruzione eseguita si ha lo spostamento di tutti i bit di una posizione.

RLC B ruota il contenuto del registro B verso sinistra; il bit b_7 va ad occupare il posto di b_0 e contemporaneamente viene ricopiato nel Carry.

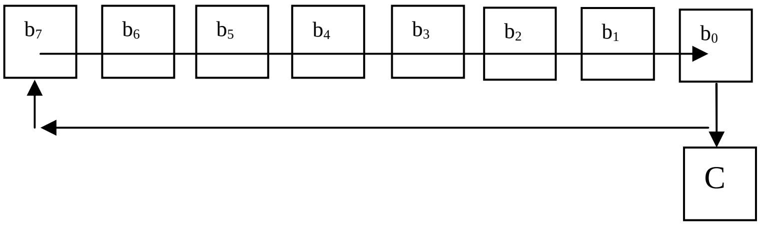


RRC (Rotate Right Circular) ovvero ***rotazione circolare verso destra*** di un dato contenuto in un registro o in una locazione di memoria la cui sintassi è:

RRC r *oppure* ***RRC (HL)***

Per ogni istruzione eseguita si ha lo spostamento di tutti i bit di una posizione.

RRC B ruota il contenuto del registro B verso destra; il bit b_0 va ad occupare il posto di b_7 e contemporaneamente viene ricopiato nel Carry.



Con queste istruzioni è possibile, per esempio, creare giochi di luci.
Il seguente segmento di programma determina l'accensione sequenziale e ciclica verso destra di uno di 8 led che costituiscono il periferico di uscita.

	LD A, 80H	;in A, b_7 alto,
LOOP:	***OUT(LED), A***	;che illumina il corrispondente led
	RRCA	;ruota verso destra tutti i bit
	Ritardo	;introduce un certo ritardo che influenza la velocità di rotazione
	JP LOOP	;e ripete.

6.9.2. Istruzioni di scorrimento

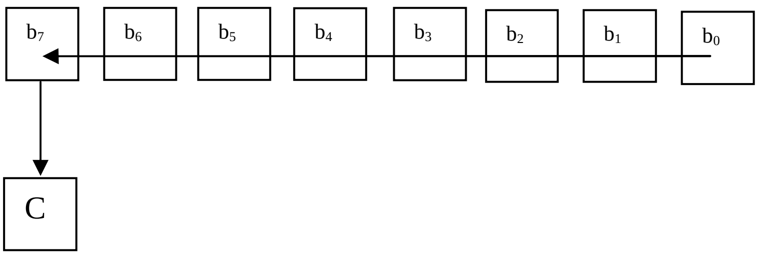
Tra le varie istruzioni di scorrimento (o di shift) possibili, esaminiamo le seguenti:

SLA (Shift Left Aritmetic) ovvero ***scorrimento verso sinistra*** di un dato contenuto in un registro o in una locazione di memoria la cui sintassi è

SLA r *oppure* ***SLA (HL)***

Per ogni istruzione eseguita si ha lo spostamento di tutti i bit di una posizione.

SLA B fa scorrere verso sinistra il contenuto del registro B; al posto di b_0 entra uno "zero" mentre il bit b_7 viene posto nel flag di C.



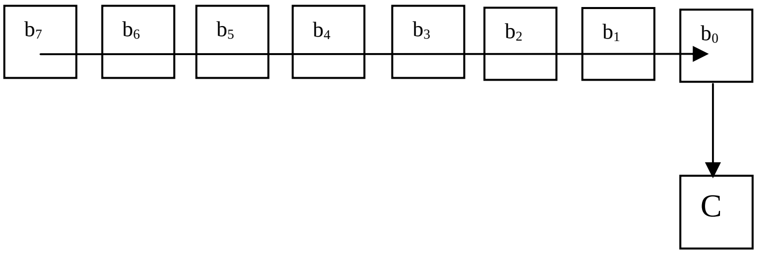
Se, in seguito ad uno scorrimento, nel registro il valore del byte si azzera, va alto il flag di Z.

SRL (Shift Rigth Logic) ovvero ***scorrimento verso destra*** di un dato contenuto in un registro o in una locazione di memoria la cui sintassi è:

SRL r *oppure* ***SRL (HL)***

Per ogni istruzione eseguita si ha lo spostamento di tutti i bit di una posizione.

SRL B ; fa scorrere verso destra il contenuto del registro B; al posto di b_7 entra uno "zero" mentre il bit b_0 viene posto nel flag di C.



Se, in seguito ad uno scorrimento, nel registro il valore del byte si azzera, va alto il flag di Z.

Facciamo la seguente considerazione. Partendo dal numero binario $(00110010)_2$ = 32H = 50_{10}, se facciamo scorrere i bit di un posto verso destra si ottiene: $(00011001)_2$ = 19H = 25_{10}: uno scorrimento verso destra equivale ad una divisione per due del numero decimale iniziale. Analogamente uno scorrimento verso sinistra equivale ad una moltiplicazione per due del numero decimale iniziale. Le istruzioni di scorrimento vengono tra l'altro utilizzate per realizzare algoritmi che consentono le operazioni di moltiplicazione e divisione.

6.10. Due istruzioni particolari

6.10.1. NOP

L'istruzione **NOP** ha una durata di 4 T_{ck} e non esegue alcuna operazione. Di solito è utilizzata nei cicli di ritardo per affinarne la durata quando non è sufficiente agire sul contenuto dei registri.

6.10.2. HALT

L'istruzione **HALT** è utilizzata per arrestare l'esecuzione del programma; per uscire dallo stato di halt, occorre resettare la CPU.

6.11. Le istruzioni di chiamata e ritorno dai sottoprogrammi

Per introdurre il concetto di ***sottoprogramma*** *(o* ***subroutine)***, e specificare le relative istruzioni di "chiamata" al sottoprogramma e "ritorno" al programma principale (***main***), facciamo riferimento all'esempio che segue.

<u>Esempio</u>: software di gestione di un circuito lampeggiatore.
Ipotizziamo che sia la linea D_0 a pilotare l'accensione e lo spegnimento del led che vogliamo far lampeggiare; sia inoltre (LED) l'indirizzo del periferico di uscita
Il segmento di programma che genera un lampeggio può essere strutturato in modo semplice con un ciclo la cui successione prevede il caricamento in A del codice di accensione, l'invio dello stesso sul led ed una routine di ritardo la cui durata determina il tempo di accensione. Analogamente si procede con la fase di spegnimento.

```
          LD SP, 0FFFFH
CICLO:    LD A, 01H         ;codice di accensione
          OUT (LED), A

          LD B, 0FFH
LOOP1A:   LD DE, 25A7H      ;inizio routine di ritardo
LOOP1:    DEC DE
          LD A, D
          OR E
```

```
              JP NZ, LOOP1
              DEC B
              JP NZ, LOOP1A      ;fine routine di ritardo

              LD A, 00H          ;codice di spegnimento
              OUT (LED),A

              LD B, 0FFH
LOOP2A:       LD DE, 25A7H       ;inizio routine di ritardo
LOOP2:        DEC DE
              LD A, D
              OR E
              JP NZ, LOOP2
              DEC B
              JP NZ, LOOP2A      ;fine routine di ritardo


              JP CICLO
              END
```

L'esempio riportato ci consente di fare la seguente osservazione.
Siccome il singolo ciclo completo di accensione e spegnimento deve prevedere due identiche routine di ritardo (riportate in grassetto) che determinano tempi uguali per l'accensione e lo spegnimento, abbiamo dovuto scrivere, in due punti differenti del programma, ***per due volte la stessa sequenza di istruzioni***. E' evidente che sarebbe più comodo scrivere tale sequenza una sola volta per poi richiamarla all'occorrenza.
Da tale osservazione nasce la comodità di ***strutturare il ciclo di ritardo come un sottoprogramma***: esso sarà un segmento di programma che, nell'ambito del programma principale verrà richiamato più volte.

6.11.1. Le istruzioni CALL e RET

Per determinare il "salto" dal main al sottoprogramma si usa l'istruzione:

CALL label

dove label è l'etichetta che identifica la subroutine.
Una volta eseguite tutte le istruzioni della subroutine si deve ritornare alla riga immediatamente successiva del main che ha comandato il salto; ciò viene eseguito dalla istruzione:

RET

che deve sempre essere l'ultima istruzione del sottoprogramma.
Alla luce di quanto detto il programma precedentemente scritto può essere compilato, in modo più compatto e leggibile, come segue:

Programma principale (main)

```
0100              LD SP, 0FFFFH
0103   CICLO:     LD A, 01H
0105              OUT (LED), A          ;accende il led
0107              CALL RITARDO          ;salta alla label RITARDO
010A              LD A, 00H
010C              OUT (LED),A           ;spegne il led
010E              CALL RITARDO          ;salta alla label RITARDO
0111              JP CICLO
```

Subroutine di ritardo

```
0114  RITARDO:  LD B, 0FFH      ;inizio routine di ritardo
      LOOP:     LD DE, 25A7H
      LOOP1:    DEC DE
                LD A, D
                OR E
                JP NZ, LOOP1
                DEC B
                JP NZ, LOOP     ;fine routine di ritardo
                RET             ;ritorna (a 010A la prima volta; a 0111 la seconda volta)

                END
```

Abbiamo riportato, in corrispondenza delle single linee di istruzioni del programma principale, gli indirizzi delle locazioni di memoria EPROM dove sono memorizzati i relativi codici operativi, per poter illustrare dettagliatamente come operano le istruzioni CALL e RET. (Come al solito, gli indirizzi partono da 0100H anche se non sono state riportate l'istruzione JP e la pseudo ORG).
Quando viene decodificata l'istruzione **CALL RITARDO**, il Program Counter (PC) contiene già l'indirizzo previsto per la successiva istruzione (**010AH**); nel prosieguo dell'esecuzione di **CALL RITARDO** si verificano contemporaneamente i seguenti due eventi:

- nell'area RAM di stack viene memorizzato l'indirizzo **010AH** che corrisponde all'istruzione **LD A, 00H,** ovvero l'indirizzo di ritorno;
- nel PC viene caricato l'indirizzo **0114H**, ovvero l'indirizzo iniziale della subroutine RITARDO.

Quando, al termine della subroutine RITARDO la CPU esegue l'istruzione **RET**, il PC si carica con l'indirizzo **010AH** precedentemente salvato nell'area di stack: in tal modo è possibile ritornare alla riga di programma immediatamente successiva a quella contenente la prima istruzione CALL.
Analogamente, decodificata la seconda istruzione **CALL RITARDO**, il Program Counter (PC) contiene già l'indirizzo previsto per la successiva istruzione (**0111H**); nel prosieguo dell'esecuzione di **CALL RITARDO** si verificano contemporaneamente i seguenti due eventi:

- nell'area RAM di stack viene memorizzato l'indirizzo **0111H** che corrisponde all'istruzione **LD A, 00H**, ovvero l'indirizzo di ritorno;
- nel PC viene caricato l'indirizzo **0114H**, ovvero l'indirizzo iniziale della subroutine RITARDO.

Come per l'istruzione JP, anche le istruzioni CALL e RET possono essere condizionate, nel senso che vengono eseguite solo al verificarsi di una condizione segnalata da un flag; il programmatore può decidere se condizionare il salto ed il ritorno in relazione al verificarsi della "condizione" o della "non condizione":

ZERO (Z = 1)	CALL Z, label	RET Z
NON ZERO (Z = 0)	CALL NZ, label	RET NZ
RIPORTO (C = 1)	CALL C, label	RET C
NON RIPORTO (C = 0)	CALL NC, label	RET NC
NEGATIVO (S = 1)	CALL M, label	RET M
POSITIVO (S = 0)	CALL P, label	RET P

Esempi:

```
CALL Z, LABEL    ; salta a label se Z = 1
CALL NZ, LABEL   ; salta a label se Z = 0
RET Z            ; ritorna se Z = 1
RET NZ           ; ritorna se Z = 0
```

L'esecuzione delle istruzioni CALL e RET incondizionate o condizionate, non modificano alcun flag.

6.11.2. Struttura di programmi con subroutines

L'esempio precedente deve essere generalizzato: per tale motivo ricordiamo quanto studiato nel paragrafo 6.4.3.2. ovvero che molto spesso, in presenza di programmi che prevedono subroutines, è indispensabile il salvataggio dei registri e dei flags. In questo caso bisogna prevedere le istruzioni PUSH e, di conseguenza, le istruzioni POP per il recupero di quanto è stato necessario salvare. Le istruzioni PUSH vengono normalmente inserite come prime istruzioni del sottoprogramma mentre quelle POP sono inserite al termine del sottoprogramma, immediatamente prima di RET.
Dunque, in generale, la struttura del software in presenza di sottoprogrammi viene organizzata nel seguente modo:

```
                Inizializzazioni
                .
                .
                Programma principale
                LD SP, nn
ciclo:          Prima istruzione……………………..
                Seconda istruzione……………………
                .
                .
                Call Label     (interrompe l'esecuzione e salta a Label)
                i-esima istruzione
                .
                .
                .
                Ultima istruzione…………………….
                JP ciclo (esegue un nuovo ciclo)

                Subroutine
Label:          PUSH AF
                PUSH BC
                .
                .
                prima istruzione subroutine
                seconda istruzione subroutine
                .
                .
                ultima istruzione subroutine
                POP BC
                POP AF
                RET (riprende dalla "i-esima istruzione")

                END
```

7. PROGRAMMI IN ASSEMBLY

7.1. Generalità

Come conclusione di questa prima parte del Corso, relativa allo studio dei concetti fondamentali della logica programmata, riportiamo alcuni esempi di programmi in Assembly che racchiudono tutte le tematiche sviluppate nei capitoli precedenti.
Nel ***paragrafo 7.2*** sono riportati segmenti di programma che rappresentano delle ***routine di base***.
Nel ***paragrafo 7.3*** sono riportati segmenti di programmi di ***lettura dati da periferici di ingresso.***
Nel ***paragrafo 7.4*** sono riportati segmenti di programmi di ***scrittura dati su periferici di uscita***.
Nel ***paragrafo 7.5*** sono riportati programmi di ***simulazione di circuiti integrati***.

Quasi tutti i programmi riportati possono essere verificati in modalità simulata utilizzando il sistema didattico Micro-Computer Emulator dMcE. Vanno aggiunte, dove occorrono, le subroutines di ritardo. La funzionalità di tutti i programmi può essere anche verificata in modalità reale con l'ausilio del sistema didattico della DENEB, disponibile nel Laboratorio di Sistemi dell'Istituto.
Le principali caratteristiche hardware del sistema DENEB sono le seguenti.

BANCO DI MEMORIA.
Realizzato con una EPROM di capacità 16 Kbyte ed una RAM di capacità 8 Kbyte.

Indirizzi:	EPROM	0000H – 3FFFH
	RAM	4000H – 5FFFH

INTERFACCE DI I/O NON PROGRAMMABILI.
Interfaccia di ingresso realizzata con il C.I. 74244 ed interfaccia di uscita realizzata con il C.I. 74374.

Indirizzi:	Interfaccia di ingresso	10H
	Interfaccia di uscita	10H

INTERFACCIA DI I/O PROGRAMMABILE.
Interfaccia PIO Z80.

Indirizzi:	PIOAD	00H
	PIOBD	01H
	PIOAC	02H
	PIOBC	03H

7.2. Le routines di base

7.2.1. Lettura, scrittura e spostamento dati in RAM

Hardware: nessuno

7.2.1.1. Scrittura di dati in RAM

Scrivere in RAM , a partire dall'indirizzo 8000H, i seguenti due dati:

dato 1 = **2FH**; dato 2 = **A7H**

a) uso della sintassi **LD (HL), nn**

```
LD HL, 8000H
LD (HL), 2FH
INC HL
LD (HL), 0A7H
HALT
END
```

b) uso della sintassi **LD (HL), r**

```
        LD HL, 8000H
        LD B, 2FH
        LD (HL), B
        INC HL
        LD B, 0A7H
        LD (HL), B
        HALT
        END
```

7.2.1.2. Lettura di dati da RAM

Leggere dalla RAM i due dati memorizzati con l'esercizio precedente.

Uso della sintassi **LD r, (HL)**

```
        LD HL, 8000H
        LD A, (HL)
        INC HL
        LD B, (HL)
        HALT
        END
```

7.2.1.3. Spostamento di un blocco di dati in RAM

Spostare 10 dati dall'indirizzo di partenza 8000H all'indirizzo di arrivo 9000H.

```
        LD HL,8000H         ;in HL indirizzo di partenza.
        LD DE,9000H         ;in DE indirizzo di arrivo.
        LD B, 0AH           ;in B numero dati da spostare.
LOOP:   LD A, (HL)          ;in A dato di origine,
        LD (DE),A           ;trasferito alla locazione di destinazione.
        INC HL              ;punta locazione successiva origine
        INC DE              ;punta locazione successiva destinazione
        DEC B               ;conta dati trasferiti decrementando B
        JP NZ, LOOP         ;se B≠0 (dati non tutti trasferiti), ripete ciclo
        HALT                ;altrimenti ferma l'esecuzione del programma.
        END
```

Quando la distanza tra l'indirizzo di partenza e quello di destinazione non supera il valore 255 (FF in esadecimale), possiamo usare in alternativa il registro IX, con una riduzione delle linee di programma. Supponiamo infatti che l'indirizzo di partenza sia 8000H e quello di destinazione 80FFH: questo significa che tra i due indirizzi c'è uno "spiazzamento" pari a FFH (255). Il programma diventa:

```
        LD IX,8000H         ;in IX indirizzo di partenza.
        LD B, 0AH           ;in B numero dati da spostare.
LOOP:   LD A, (IX)          ;in A dato di origine,
        LD (IX + 0FFH),A    ;trasferito alla locazione di destinazione.
        INC IX              ;punta locazione successiva origine
        DEC B               ;conta dati trasferiti decrementando B
        JP NZ, LOOP         ;se B≠0 (dati non tutti trasferiti), ripete ciclo
        HALT                ;altrimenti ferma l'esecuzione del programma.
        END
```

7.2.2. Lettura da ingresso e memorizzazione dati in RAM

Hardware: solo interfaccia di ingresso

7.2.2.1. Acquisizione continua di dati e loro memorizzazione in RAM

```
        LD HL, 8000H        ;in HL 1° indirizzo RAM
LOOP:   IN A, (PERIN)       ;acquisisce dato
        LD (HL), A          ;e lo memorizza
        INC HL              ;punta locazione successiva
        JP LOOP             ;ripete ciclo
        END
```

7.2.2.2. Acquisizione di un numero prefissato di dati e memorizzazione in RAM

```
        LD HL 8000H         ;in HL 1° indirizzo RAM
        LD B, 0CH           ;inizializza registro contatore dati (12)
LOOP:   IN A, (PERIN)       ;acquisisce dato
        LD (HL),A           ;e lo memorizza all'indirizzo di destinazione.
        INC HL              ;punta la locazione successiva
        DEC B               ;conta dati acquisiti decrementando B
        JP NZ, LOOP         ;se B≠0 (dati non tutti acquisiti), ripete ciclo
        HALT                ;altrimenti ferma l'esecuzione del programma.
        END
```

7.2.3. Lettura da RAM e scrittura su uscita

Hardware: solo interfaccia di uscita

7.2.3.1. Lettura continua di dati da RAM e loro pubblicazione

```
        LD HL, 8000H        ;in HL 1° indirizzo RAM
LOOP:   LD A, (HL)          ;in A dato letto dalla memoria
        OUT (PEROUT),A      ;scritto in uscita
        CALL RITARDO        ;per un certo tempo (pausa).
        INC HL              ;punta locazione successiva
        JP LOOP             ;ripete ciclo
        END
```

7.2.3.2. Lettura da RAM di un numero prefissato di dati e loro pubblicazione

```
        LD HL,8000H         ;in HL 1° indirizzo RAM
        LD B, 0CH           ;in B numero dati da leggere (12)
LOOP:   LD A,(HL)           ;in A dato letto dalla memoria
        OUT (PEROUT),A      ;scritto in uscita
        CALL RITARDO        ;per un certo tempo (pausa).
        INC HL              ;punta locazione successiva
        DEC B               ;conta numero dati ancora da leggere
        JP NZ, LOOP         ;se B≠0 (dati non tutti letti), ripete ciclo
        HALT                ;altrimenti ferma l'esecuzione del programma.
        END
```

7.3. Esempi di lettura dati da periferici di ingresso

7.3.1. Rilievo e conteggio di impulsi positivi

Il programma potrebbe essere una subroutine in grado di rilevare e contare passaggi di oggetti attraverso una barriera ad infrarossi.
Hardware: si ipotizza che gli impulsi di uscita del sistema fotoaccoppiatore siano disponibili sulla linea D_0 dell'interfaccia di ingresso.

```
        LD B, 00H         ;inizializza registro contatore degli impulsi
LIVL:   IN A, (PERIN)     ;legge il dato (un intero byte)
        BIT 0, A          ;testa D0
        JP Z, LIVL        ;finché D0 è basso, resta bloccato nella lettura del livello basso
LIVH:   IN A, (PERIN)     ;quando D0 diventa alto avanza e rilegge il dato (un intero byte)
        BIT 0, A          ;testa D0
        JP NZ, LIVH       ;finché D0 è alto, resta bloccato nella lettura del livello alto
        INC B             ;quando D0 ritorna basso, conta l'impulso
        JP LIVL           ;e ripete
        END
```

7.3.2. Rilievo dello stato di un sensore

Il programma tiene sotto controllo lo stato di un sensore (ad esempio un sensore di livello) la cui uscita è, per ipotesi, normalmente bassa; quando il sensore si attiva, manda alta la corrispondente linea. Questo evento viene rilevato per comandare una generica subroutine di servizio.
Hardware: solo interfaccia di ingresso; si testa la linea D_4

```
        LD SP, 0FFFFH     ;fissa base dello stack
LEGGI:  IN A, (PERIN)     ;legge ingresso
        AND 08H           ;isola D4
        JP Z, LEGGI       ;ripete la lettura finché D4 resta basso.
        CALL SERVIZIO     ;se D4 diventa alto, salta a "SERVIZIO"
        END
```

7.3.3. Rilievo della modifica dello stato di uno o più sensori

Hardware: interfaccia di ingresso e di uscita

```
        OUT (PEROUT), 00H ;spegne i led.
LOOP:   IN A, (PERIN)     ;acquisisce primo dato
        LD B, A           ;e lo salva in B.
        CALL RITARDO      ;dopo un tempo prefissato,
        IN A, (PERIN)     ;acquisisce secondo dato
        XOR B             ;lo confronta con il primo e pone alti i bit
                          ; modificati
        OUT (PEROUT), A   ;li visualizza
        CALL RITARDO1     ;per un certo tempo
        JP LOOP           ;e ripete il ciclo
        END
```

7.4. Esempi di scrittura dati su periferici di uscita

7.4.1. Generazione via software di segnali

7.4.1.1. Generazione di un impulso negativo

Hardware: solo interfaccia di uscita; l'impulso viene prelevato sulla linea D_0

```
          SET 0, A             ;Setta D0,
          OUT (PEROUT), A      ;scritto in uscita
          CALL RIT1            ;per il tempo RIT1 sufficientemente
                               ;lungo (valore logico normalmente alto).
          RES 0, A             ;resetta D0
          OUT (PEROUT), A      ;scritto in uscita
          CALL RIT2            ;per il tempo RIT2 = durata impulso.
          SET 0, A             ;Setta di nuovo D0
          OUT (PEROUT), A      ;scritto in uscita, che resta alta.
          HALT
          END
```

7.4.1.2. Generazione di un'onda quadra (Segale di CK)

Hardware: solo interfaccia di uscita; il segnale viene prelevato sulla linea D_7

```
LOOP:     SET 7, A             ;Setta D7
          OUT (PEROUT), A      ;scritto in uscita
          CALL RIT             ;per un tempo RIT = semiperiodo H
          RES 7, A             ;resetta D7
          OUT (PEROUT), A      ;scritto in uscita
          CALL RIT             ;per un tempo RIT = semiperiodo L
          JP LOOP              ;ripete per il nuovo impulso.
          END
```

Nota: con lo stesso segmento di programma si possono generare **onde rettangolari** con valori di Duty Cycle prefissati, prevedendo due routines di ritardo diverse, dimensionate in funzione del Duty Cycle desiderato.

7.4.2. Pilotaggio di dispositivi di OUT

7.4.2.1. Pilotaggio di un MPX a due ingressi di selezione

Hardware: solo interfaccia di uscita; si utilizzano le linee D_1 e D_0 per pilotare le linee di selezione S_1 ed S_0 del MPX per un solo ciclo.

```
          LD SP, 5FFFH         ;fissa il fondo dello stack
          LD B,04H             ;in B numero di combinazioni
          LD A,00H             ;in A codice di selezione prima linea
LOOP:     OUT (MPX), A         ;selezione linea su MPX
          CALL RIT             ;pausa
          INC A                ;in A codice di selezione linea successiva
          DEC B                ;decrementa B
          JP NZ, LOOP          ;se B≠0 ripete
          HALT                 ;altrimenti finisce
          END

RIT:      LD C, 02H
LOOP1:    DEC C
          JP NZ, LOOP1
          RET
```

7.4.2.2. Pilotaggio multiplexato di un visualizzatore a due cifre con display a 7 segmenti

Hardware: interfaccia di ingresso e di uscita. Le linee di uscita $D_0 – D_3$ del bus dati pilotano la decodifica BCD/7segmenti mentre le linee D_6 e D_7 pilotano due BJT funzionanti in ON/OFF attraverso i quali sono collegati a massa i catodi dei display, rispettivamente, delle unità e delle decine. Il programma acquisisce il numero da visualizzare dal periferico di ingresso; bisogna pertanto impostare su di esso detto numero in codice BCD.

```
        LD SP, 5FFFH        ;fissa il fondo dello stack
LOOP:   IN A, (PERIN)       ;legge dato da visualizzare
        PUSH AF             ;lo salva
        AND 0FH             ;isola nibble unità
        OR 40H              ;attiva D6, ovvero display unità
        OUT (DISP), A       ;e visualizza unità
        POP AF              ;recupera contenuto iniziale
        SRLA                ;4 scorrimenti verso destra
        SRLA                ;posizionano i 4 bit delle
        SRLA                ;decine, sulle linee D0 – D3
        SRLA                ;per la visualizzazione.
        OR 80H              ;attiva D7 ovvero display decine
        OUT (DISP), A       ;e visualizza decine
        JP LOOP             ;ripete ciclo di lettura e visualizzazione
        END
```

7.4.2.3. Centralina per il controllo di un semaforo stradale

Hardware: solo interfaccia di uscita; si utilizzano le linee D_0 per pilotare il ROSSO, D_1 per pilotare il GIALLO e D_2 per pilotare il VERDE.

```
LOOP:   LD A, 01H           ;in A codice accensione ROSSO
        OUT (PEROUT), A     ;pubblicato in uscita
        CALL RIT            ;per un tempo RIT
        LD A, 04H           ;in A codice accensione VERDE
        OUT (PEROUT), A     ;pubblicato in uscita
        CALL RIT            ;per un tempo RIT
        LD A, 06H           ;in A codice accensione V/GIALLO
        OUT (PEROUT), A     ;pubblicato in uscita
        CALL RIT1           ;per un tempo RIT1 < RIT
        JP LOOP             ;ripete ciclo
        END
```

7.4.2.4. Centralina per gioco di luci con diodi led

Hardware: solo interfaccia di uscita.

Il programma prevede un gioco di luci composto da tre distinte fasi, realizzate con altrettante subroutines, che ciclicamente si ripetono. La prima subroutine parte con il codice di accensione CCH = 11001100, quindi con due coppie di led accesi ed effettua dodici scorrimenti verso destra della combinazione iniziale. La seconda subroutine parte con il codice di accensione AAH = 10101010, quindi con una sequenza di led accesi e spenti ed effettua dodici scorrimenti verso sinistra della combinazione iniziale. La terza subroutine parte con il codice di accensione FFH = 11111111, quindi con tutti i led accesi, ed esegue una successione di sei accensioni e sei spegnimenti di tutti i led. La velocità degli scorrimenti e della successione di accensione e spegnimento è modificabile agendo sulle subroutines di ritardo. Il programma evidenzia come, all'interno di un sottoprogramma (LOOP1, LOOP2 e LOOP3) si possono chiamare altri sottoprogrammi (RIT)

```
            LD SP, 5FFFH    ; fissa il fondo dello stack
            LD B, 0CCH      ; in B 1° codice accensione
            LD C, 0AAH      ; in C 2° codice accensione
            LD D, 0FFH      ; in D 3° codice accensione

;programma principale:
RIPETI:     CALL LOOP1
            CALL LOOP2
            CALL LOOP3
            JP RIPETI

LOOP1:      LD E, 0CH               ; subroutine per 12 scorrimenti a destra
            LD A,B
CICLO1:     OUT (10H), A
            CALL RIT
            RRC A
            DEC E
            JP NZ, CICLO1
            RET

LOOP2:      LD E, 0CH               ; subroutine per 12 scorrimenti a sinistra
            LD A,C
CICLO2:     OUT (10H), A
            CALL RIT
            RLC A
            DEC E
            JP NZ, CICLO2
            RET

LOOP3:      LD E, 06H               ; subroutine per sei lampeggi
            LD A,D
CICLO3:     OUT (10H), A
            CALL RIT
            CPL
            DEC E
            JP NZ, CICLO3
            RET

RIT:        PUSH AF                 ; subroutine di ritardo
            LD HL, 0FFFFH
RIP:        DEC HL
            LD A,H
            OR L
            JP NZ, RIP
            POP AF
            RET

            END
```

7.5. Simulazione di C.I.

7.5.1. Circuito sommatore tra due numeri a 4 bit.

Il programma simula il funzionamento di un circuito sommatore 74LS83.
I 4 + 4 ingressi del sommatore corrispondono fisicamente alle 8 linee di ingresso del periferico di ingresso mentre le sue 5 uscite (4 linee di uscita + riporto) corrispondono fisicamente alle corrispondenti linee meno significative del periferico di uscita.
I due numeri tra i quali si vuole fare la somma vengono presentati sul periferico di ingresso come un unico byte così come mostrato nell'esempio:
$N_1 = 1000_2$; $N_2 = 1100_2$;

1	0	0	0	1	1	0	0

I quattro bit più significativi $D_7 – D_4$ identificano N_1 mentre i quattro bit meno significativi $D_3 – D_0$ identificano N_2. Prima dell'operazione di somma si deve procedere al corretto posizionamento nei registri B ed A dei due numeri N_1 ed N_2.
Il loop legge continuamente il dato in ingresso ed aggiorna l'uscita in relazione ai dati presentati. Il risultato viene letto sulle 5 linee $D_0 – D_4$ del periferico di uscita con la precisazione che il bit D_4 rappresenta l'eventuale riporto.

```
            LD SP, 5FFFH        ;fissa il fondo dello stack
START:      IN A, (10H)         ;in A il byte contenente N1 ed N2
            PUSH AF             ;salva il contenuto di A
            AND 0FH             ;isola N2
            LD B, A             ;e lo salva in B
            POP AF              ;ripristina il contenuto di A
            SRL A
            SRL A
            SRL A               ;4 scorrimenti verso destra posizionano
            SRL A               ;correttamente N1 in A
            ADD A, B            ;somma N1 ad N2
            OUT (10H), A        ;pubblica il risultato
            JP START            ;salta per una nuova somma
            END
```

7.5.2. Circuito comparatore tra due numeri a 4 bit.

Il programma simula il funzionamento di un circuito comparatore 74LS85.
I 4 + 4 ingressi del circuito comparatore corrispondono fisicamente alle 8 linee del periferico di ingresso mentre le sue 3 uscite corrispondono fisicamente alle linee meno significative del periferico di uscita. Le condizioni: N_1<N2 ; $N_1=N_2$ ed $N_1>N_2$ vengono visualizzate, rispettivamente, sulle linee D_2, D_1 e D_0. La prima parte del programma è identica a quella relativa al circuito sommatore. La seconda parte del programma testa i flags di Z e di C che determinano salti condizionati ad altrettante semplicissime subroutines che provvedono ad attivare le linee di uscita interessate. Siccome la condizione $N_1>N_2$ non è segnalata da alcun flag, il programma prevede per essa un salto incondizionato posto dopo i due salti condizionati. In altre parole se i flag testati non si attivano, determinando i primi due salti ad U_1 ed U_2, viene necessariamente eseguita, per esclusione, la subroutine U_0 relativa alla condizione $N_1>N_2$.

```
            LD SP, 5FFFH        ;fissa il fondo dello stack
INPUT:      IN A, (10H)         ;in A il byte contenente N1 ed N2
            PUSH AF             ;salva il contenuto di A
            AND 0FH             ;isola N2
```

```
                LD B, A          ;e lo salva in B
                POP AF           ;ripristina il contenuto di A
                SRL A
                SRL A
                SRL A            ;4 scorrimenti verso destra
                SRL A            ;posizionano correttamente N1 in A
                CP B             ;confronta N1 con N2
                JP Z, U1         ;se Z=1, N1= N2 per cui salta ad U1
                JP C, U2         ;se C=1, N1< N2 per cui salta ad U2
                JP U0            ;se N1> N2 salta ad U0

U0:             PUSH AF          ;subroutine di accensione di D0
                LD A, 01H
                OUT (10H), A
                POP AF
                JP INPUT

U1:             PUSH AF          ;subroutine di accensione di D1
                LD A, 02H
                OUT (10H), A
                POP AF
                JP INPUT

U2:             PUSH AF          ;subroutine di accensione di D2
                LD A, 04H
                OUT (10H), A
                POP AF
                JP INPUT

                END
```

7.5.3. Circuito contatore M = 10

Il programma simula il funzionamento di un circuito contatore in avanti modulo 10 74LS90.
L'ingresso di CK corrisponde fisicamente alla linea D_0 del periferico di ingresso mentre le quattro uscite vengono prelevate sulle linee D_0-D_3 del periferico di uscita. Gli impulsi vengono generati da un pulsante; il conteggio degli impulsi può essere visualizzato da quattro led o da un display a 7 segmenti con relativa decodifica.
Il programma testa in continuazione la linea di clock e per ogni impulso rilevato aggiorna le uscite che visualizzano, in binario, il corrispondente numero.

```
                LD SP, 5FFFH          ;fissa il fondo dello stack
                LD B, 00H             ;inizializza B
LOOP:           LD A, 00H             ;inizializza A
LOOP1:          OUT (10H), A          ;visualizza il conteggio
                CALL CLOCK            ;salta a CLOCK per leggere gli impulsi
                LD A, B               ;aggiorna il numero di impulsi letti
                CP 0AH                ;confronta con 10
                JP Z, LOOP            ;se il numero di impulsi letti ≠ 10, prosegue,
                JP LOOP1              ;e salta per leggere il nuovo impulso

CLOCK:          PUSH AF               ;subroutine per il rilievo degli impulsi
LIVL:           IN A, (10H)           ;acquisisce il dato
                BIT 0, A              ;testa D0
```

```
                JP Z, LIVL          ;finché D0 è basso, resta bloccato nella lettura
                                    ;del livello basso
LIVH:           IN A, (10H)         ;acquisisce il dato
                BIT 0, A            ;testa D0
                JP NZ, LIVH         ;finché D0 è alto, resta bloccato nella lettura del
                                    livello alto
                INC B               ;quando D0 ritorna basso, incrementa il contatore
                POP AF
                RET

                END
```