# Università degli Studi di Milano

**Corso di Laurea in**
**Sicurezza dei Sistemi e delle Reti Informatiche**

# Lezione 2 – Struttura e set di istruzioni della CPU LC-2

**NELLO SCARABOTTOLO**

**Architetture e reti logiche**

Modulo 6 - Unità Didattica 2

# Indice

# 1. CPU LC-2

## 1.1 Struttura della CPU LC-2

La CPU didattica LC-2 – utilizzata in questo modulo dell'insegnamento di Architetture e reti logiche per illustrare la programmazione in linguaggio macchina – ha la struttura interna raffigurata in Figura 6.2.2-1.

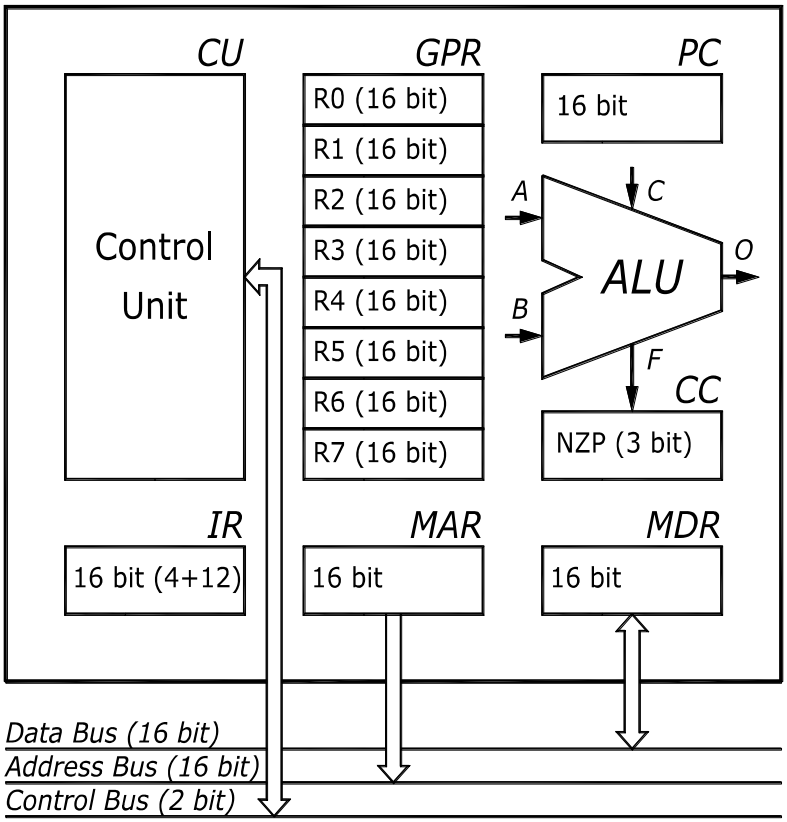



**Figura 6.2.2-1 — Struttura interna della CPU LC-2**

Si tratta di una CPU con spazio di indirizzamento di $2^{16}$ celle (come evidenziato dalla dimensione del registro ***MAR***: *Memory Address Register*) ciascuna contenente una parola di memoria da 16 bit (come evidenziato dalla dimensione del registro ***MDR***: *Memory Data Register*).

La CPU LC-2 è una macchina RISC: il suo ***ISA*** (*Instruction Set Architecture*) è infatti costituito da istruzioni macchina che occupano ciascuna una sola cella di memoria (istruzioni a 16 bit, da cui la dimensione del registro ***IR***: *Instruction Register*) e con codice operativo (*opcode*) di soli 4 bit. Ciò consente dunque di codificare solo $2^4$=16 diverse istruzioni macchina.

L'approccio RISC è evidente anche nell'adozione di un numero limitato di modi di indirizzamento:

**immediato** l'operando è contenuto nell'istruzione macchina;

**diretto** l'istruzione macchina fornisce l'indirizzo della cella di memoria contenente l'operando;

**indiretto** l'istruzione macchina fornisce l'indirizzo di una cella di memoria contenente a sua volta l'indirizzo della cella di memoria contenente l'operando;

**base+offset** l'istruzione macchina indica un registro GPR e un valore numerico da sommare al contenuto del registro GPR per ottenere l'indirizzo della cella di memoria contenente l'operando.

I registri di lavoro (***GPR***: *General Purpose Registers*) della CPU LC-2 sono 8, denominati da **R0** a **R7**.

Il registro ***CC*** (*Condition Codes*) è costituito da 3 bistabili – denominati **N**, **Z** e **P** – che vengono aggiornati ogni volta che un'istruzione macchina modifica il contenuto di uno dei GPR. L'aggiornamento consiste nel portare a uno N, Z o P a seconda che il valore scritto nel GPR (considerato come numero binario in complemento a 2) sia negativo, nullo o positivo.

L'unità aritmetico-logica (***ALU***: *Arithmetic-Logic Unit*) è estremamente semplice e svolge solo le operazioni strettamente indispensabili: ADD, AND, NOT. Qualora necessario, si deve ricorrere a opportuni algoritmi per eseguire le operazioni mancanti (per es., utilizzare il teorema di De Morgan per svolgere somme logiche OR).

## 1.2 Linguaggio macchina della CPU LC-2

L'ISA della CPU LC-2 è schematicamente riassunto in Figura 6.2.2-2.

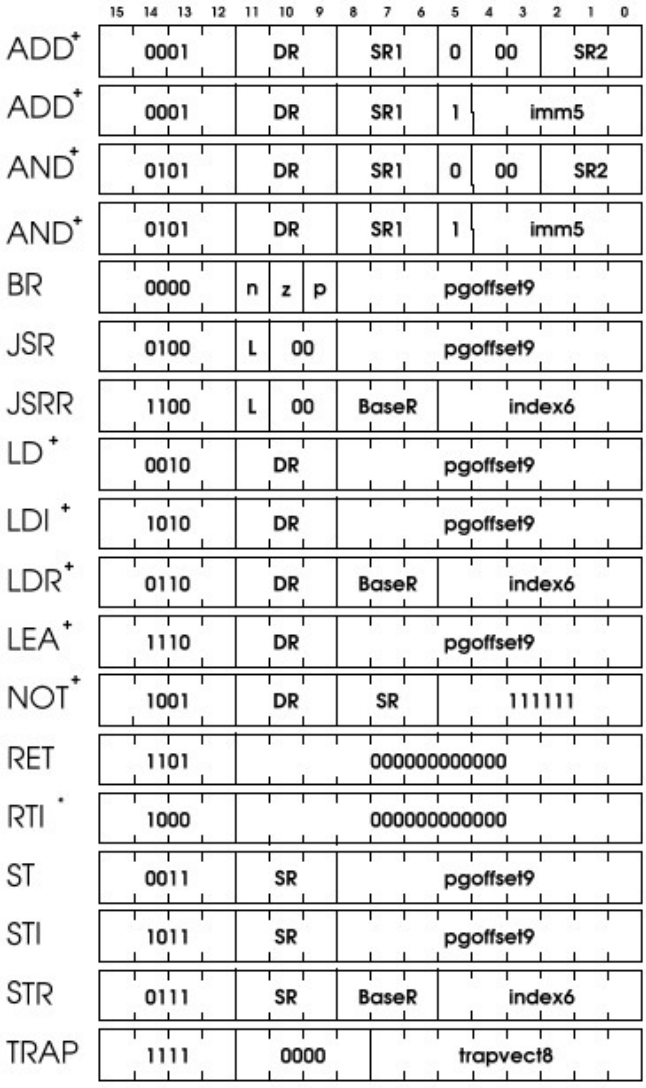



**Figura 6.2.2-2 — ISA della CPU LC-2**

Come si può vedere, i primi 4 bit di ogni istruzione sono dedicati a specificare il codice operativo (*opcode*) mentre i rimanenti 12 bit servono a reperire gli operandi.

L'*opcode* è lungo 4 bit poiché l'ISA della CPU LC-2 è costituita da 16 istruzioni; pertanto i 4 bit più significativi di un'istruzione indicano, in binario, quale istruzione deve essere eseguita. A questo punto, definito il tipo di operazione, rimangono 12 bit per gli operandi: come vengono assegnati questi bit dipende essenzialmente dall'istruzione che si sta eseguendo e dal modo di indirizzamento.

Per ogni istruzione, vengono ora riportati in dettaglio la codifica, il funzionamento, il modo di costruzione degli eventuali operandi.

### 1.2.1 Istruzioni operative

***Istruzione ADD: somma aritmetica***

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **ADD** | **0** | **0** | **0** | **1** | **DR** | | | **SR1** | | | **0** | × | × | **SR2** | | |

**SR1 + SR2 → DR**

Porta nel registro destinazione DR la somma dei numeri a 16 bit contenuti nei due registri sorgente SR1 e SR2.

Nei registri DR, SR1 e SR2 è contenuta una "parola" da 16 bit: infatti tali registri sono proprio i registri GPR. Dal momento che la CPU LC-2 ha 8 registri GPR (da R0 a R7), la codifica binaria del registro da utilizzare come sorgente o destinazione richiede appunto 3 bit. Gli zeri tra SR1 e SR2 sono quindi un riempitivo.

Esempio: **ADD R0, R1, R2** ;somma il contenuto di R1 e R2 e salvalo in R0.

R1 = x3001 → 0011 0000 0000 0001

R2 = x0005 → 0000 0000 0000 0101

R0 = x3001 + x0005 = x3006 → 0011 0000 0000 0110

***Istruzione ADD: somma aritmetica***

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **ADD** | **0** | **0** | **0** | **1** | **DR** | | | **SR** | | | **1** | **imm5** | | | | |

**SR + $imm5_{16}$ → DR**

Porta nel registro destinazione DR la somma del numero a 16 bit contenuto nel registro sorgente SR e del numero a 16 bit ottenuto come estensione con segno della costante a 5 bit in complemento a 2 **imm5** presente nell'istruzione.

Ricordando che per estendere il segno di un numero in complemento a 2 si deve ripetere il suo bit più significativo, tale estensione si comporta nel modo seguente:

se imm5 ≥ 0 (bit più significativo di imm5 = 0) $imm5_{16}$ = 0000 0000 000 imm5

se imm5 < 0 (bit più significativo di imm5 = 1) $imm5_{16}$ = 1111 1111 111 imm5

In questo caso quello che viene sommato non è il contenuto di due registri come nel caso precedente, ma al contenuto di un registro sorgente SR viene sommata una costante numerica: il risultato di questa addizione viene salvato nel registro destinazione DR.

Il problema risiede però nel fatto che i registri contengono parole da 16 bit, mentre la costante da sommare ha uno spazio ridotto, solo 5 bit. Dal momento che non è possibile

sommare parole con lunghezza differente, occorre estendere la costante numerica in modo che venga intesa come una parola lunga 16 bit.

Ricordiamo brevemente come funziona il **complemento a 2**:

- se si hanno a disposizione *n* bit la rappresentazione in complemento a 2 permette di rappresentare numeri **interi positivi** da 0 a $2^{n-1}-1$ e numeri **interi negativi** da -1 a $-2^{n-1}$. Nel nostro esempio abbiamo uno spazio di 5 bit, quindi potremo rappresentare in complemento a 2 i numeri **interi** che vanno da 0 a 15 (**positivi**) e da -1 a -16 (**negativi**). Il bit più significativo serve come **segno**: i numeri positivi hanno il bit più significativo a 0, i numeri negativi hanno il bit più significativo a 1.
- la regola da seguire per portare un numero in complemento a 2 è la seguente:
  - se il numero è positivo, il suo complemento a due è il numero stesso;
  - se il numero negativo, il suo complemento a due è il suo modulo negato e quindi sommato a uno.
- per esempio, se vogliamo rappresentare in complemento a 2 il numero decimale 14, il risultato è dato dalla semplice conversione in binario (su 5 bit):
  - 14 → 01110 (il bit più significativo a 0 sta ad indicare che il numero rappresentato è POSITIVO)
- supponiamo ora di voler rappresentare, sempre in complemento a 2, il numero decimale -14; il risultato è il seguente (su 5 bit):
  - non considero il segno è scrivo il numero in binario su 5 bit: 14 → 01110
  - inverto tutti i bit: 01110 → 10001
  - sommo 1 al risultato: -14 → 10010 (il bit più significativo a 1 sta ad indicare che il numero rappresentato è NEGATIVO).

Tornando alla CPU LC-2, la costante numerica che può essere sommata al contenuto di un registro sorgente può valere da -16 a +15. Le precedenti considerazioni sul complemento a 2 dovrebbero chiarire il fatto che, da una parola a 5 bit (la costante da sommare), la sua estensione in 16 bit comporta esclusivamente la ripetizione del suo bit più significativo: 1 se il numero in complemento a 2 è negativo, 0 se è positivo.

Esempio: **ADD R0, R1, #-5** ;somma il contenuto di R1 e -5 e salvalo in R0.

R1 = x3001 → 0011 0000 0000 0001 (in decimale 12289)

imm5 = #-5 → $imm5_{16}$ = 1111 1111 1111 1011

R0 = x2FFC (in decimale 12284)

***Istruzione AND: prodotto logico***

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **AND** | **0** | **1** | **0** | **1** | **DR** | | | **SR1** | | | **0** | × | × | **SR2** | | |

**SR1 AND SR2 → DR**

Porta nel registro destinazione DR il prodotto logico bit a bit dei numeri a 16 bit contenuti nei due registri sorgente SR1 e SR2.

In questo caso valgono le considerazioni fatte per l'ADD.

Esempio: **AND R0, R1, R2** ;esegui il prodotto logico bit a bit del contenuto di R1 e R2 e salvalo in R0.

R1 = x3012 → 0011 0000 0001 0010

R2 = x2112 → 0010 0001 0001 0010

R0 = x2212 → 0010 0000 0001 0010

***Istruzione AND: prodotto logico***

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **AND** | **0** | **1** | **0** | **1** | **DR** | | | **SR** | | | **1** | **imm5** | | | | |

**SR AND $|imm5_{16}| \rightarrow$ DR**

Porta nel registro destinazione DR il prodotto logico bit a bit del numero a 16 bit contenuto nel registro sorgente SR e del numero a 16 bit ottenuto come estensione in valore assoluto della costante a 5 bit **imm5** presente nell'istruzione. Dunque:

$|imm5_{16}|$ = 0000 0000 000 imm5

Diversamente dal caso descritto dall'istruzione ADD in questo caso la costante numerica è espressa in **valore assoluto**, pertanto l'unica cosa da fare è aggiungere una serie di zeri alla costante numerica, espressa in valore assoluto, fino ad arrivare ad una parola da 16 bit.

Esempio: **AND R0, R1, #8** ;esegui il prodotto logico del numero contenuto in R1 e 8 quindi salvalo in R0.

R1 = x301A → 0011 0000 0001 1010

$|imm5_{16}|$ = x0008 → 0000 0000 0000 1000

R0 = x0008 → 0000 0000 0000 1000

***Istruzione NOT: negazione logica***

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **NOT** | **1** | **0** | **0** | **1** | **DR** | | | **SR** | | | × | × | × | × | × | × |

**$\overline{SR} \rightarrow$ DR**

Porta nel registro destinazione DR la negazione bit a bit del numero a 16 bit contenuto nel registro sorgente SR.

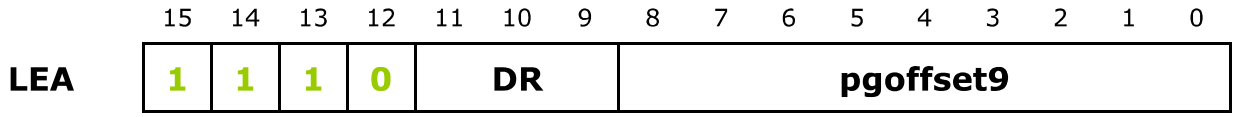
Esempio:

SR = x3004 → 0011 0000 0000 0100

DR = xCFFB → 1100 1111 1111 1011

### 1.2.2 Istruzioni di trasferimento

***Istruzione LEA (Load Effective Address): caricamento indirizzo***




**ind16 → DR** **modalità di indirizzamento Immediato**

Per indirizzare la memoria, serve costruire un indirizzo a 16 bit, ma nell'istruzione – tolto codice operativo e indicazione del registro destinazione – sono disponibili solo 9 bit.

Una tipica soluzione delle macchine RISC è quella di concatenare i bit più significativi del PC (***MS***: *MOst Significant*, 7 in questo caso) con i bit disponibili nell'istruzione, e costruire un indirizzo che si riferisce a una delle celle dalla "pagina" di memoria corrente, nella quale si trova il programma in esecuzione. Se indichiamo con **&** l'operatore di concatenazione di stringhe di bit, l'indirizzo a 16 bit viene quindi costruito come:

ind16 = $PC_{MS7}$ & pgoffset9

Questa istruzione, oltre al codice operativo che la identifica, contiene 3 bit per il registro destinazione (uno tra R0..R7) e 9 bit per il page offset.

L'espressione **ind16 → DR** sta ad indicare che vogliamo caricare nel registro destinazione DR un indirizzo a 16 bit. Purtroppo però, per indicare questo indirizzo a 16 bit, abbiamo a disposizione solo per 9 bit (il page offset). Dobbiamo quindi "recuperare" altri 7 bit da un'altra parte per arrivare all'indirizzo da 16 bit da caricare. Questi 7 bit vengono presi dai 7 bit più significativi del Program Counter (in posizione corrente), e saranno i bit più significativi dell'indirizzo a 16 bit.

Volendo trovare un'analogia "letteraria" si potrebbe considerare ind16 come un libro, i cui numeri di pagina sono dati da $PC_{MS7}$. Pgoffset9 indica una riga di una singola pagina.

*Esempio:* ***LEA R1, x30F4*** *;vorrei caricare x30F4 in R1*

L'istruzione scritta in questo modo genera un errore da parte dell'assembler, che non è in grado di inserire l'indirizzo x30F4 nei 9 bit disponibili. Per ottenere il risultato desiderato, è necessario quindi che $PC_{MS7}$ contengano già la sequenza di bit 0011 000 (x30) per poter scrivere l'istruzione (comprensibile all'assembler):

**LEA R1, xF4** ;carico x30F4 in R1

Questa istruzione LEA viene codificata come: 1110 001 **011110100**

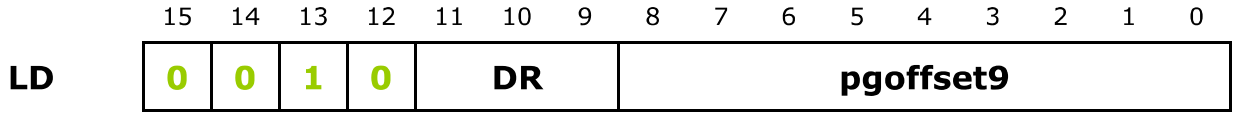
- 1110: opcode dell'istruzione LEA
- 001: registro destinazione
- ***011110100***: page offset

PC = x3023 = **0011 000**0 0010 0011 (i 7 bit più significativi in grassetto)

**0011 000** & ***011110100*** (MS7 concatenato con page offset) = x30F4

Nel resto della dispensa, per comodità di lettura, quando si farà riferimento a istruzioni che devono costruire un indirizzo a 16 bit si utilizzerà la forma "scorretta" per l'assembler con i 16 bit esplicitati come 4 cifre esadecimali.

***Istruzione LD (Load): caricamento diretto da memoria***




**M(ind16) → DR** **modalità di indirizzamento Diretto**

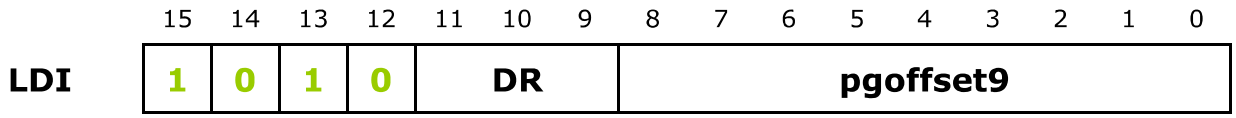
L'istruzione porta nel registro destinazione DR il contenuto della cella di indirizzo ind16. Serve dunque a leggere il contenuto di una cella situata nella pagina corrente (nella quale si trova il programma in esecuzione).

Il meccanismo per la costruzione dell'indirizzo a 16 bit mediante page offset è lo stesso descritto in precedenza. In questo caso però, nel registro destinazione non viene caricato l'indirizzo ind16, ma il contenuto della cella che ha come indirizzo ind16.

Esempio: **LD R1, x3002**

- con l'istruzione **LEA** alla fine mi troverei ad avere **R1 = x3002**;
- supponiamo che in corrispondenza dell'indirizzo x3002 ci sia la parola x0005. Con l'istruzione **LD** alla fine mi trovo ad avere **R1 = x0005**.
- in pratica **LEA** carica un indirizzo, **LD** carica un dato.

***Istruzione LDI (Load Indirect): caricamento indiretto da memoria***




**M(M(ind16)) → DR** **modalità di indirizzamento Indiretto**

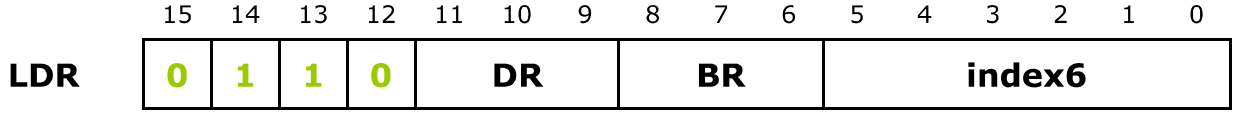
L'istruzione porta nel registro destinazione DR il contenuto della cella il cui indirizzo è contenuto nella cella di memoria di indirizzo ind16. Serve dunque a leggere il contenuto di una cella situata in qualsiasi posizione dello spazio di indirizzamento, ma richiede due accessi a memoria (uno per leggere l'indirizzo, l'altro per leggere il dato).

In questo caso facciamo direttamente un esempio, partendo dalle seguenti ipotesi:

- vogliamo eseguire l'istruzione **LDI R1, xE231**
- in **xE231** è contenuta la parola **x4B21**
- in **x4B21** è contenuta la parola **x213A**

Date queste ipotesi l'istruzione **LDI** porterà la parola **x213A** in **R1**.

***Istruzione LDR (Load using Register): caricamento da memoria con registro base***




**M(BR + |index6$_{16}$|) → DR** **modalità di indirizzamento Base+offset**

L'istruzione porta nel registro destinazione DR il contenuto della cella il cui indirizzo viene calcolato sommando al contenuto del registro base BR il valore a 16 bit ottenuto come estensione in valore assoluto della costante a 6 bit **index6** presente nell'istruzione:

|index6$_{16}$| = 0000 0000 00 index6

Serve dunque a leggere il contenuto di una cella situata in qualsiasi posizione dello spazio di indirizzamento, con un solo accesso a memoria.

Anche in questo caso facciamo un esempio pratico:

- vogliamo eseguire l'istruzione **LDR R0, R1, #2**
- **R1** = x3001 → 0011 0000 0000 0001
- **#2** → 000010 in 6 bit… portato a 16 bit → x0002 = 0000 0000 0000 0010
- **R1 + #2** → 0011 0000 0000 0011 → x3003
- la cella di indirizzo **x3003** contiene la parola **x1234**
- nel registro destinazione verrà salvato **x1234** (**R0 = x1234**)

***Istruzione ST (Store): scrittura diretta in memoria***

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **ST** | **0** | **0** | **1** | **1** | **SR** | | | **pgoffset9** | | | | | | | | |

**SR → M(ind16)** **modalità di indirizzamento Diretto**

Duale di LD, l'istruzione porta il contenuto del registro sorgente SR nella cella di indirizzo ind16. Serve dunque a scrivere il contenuto di una cella situata nella pagina corrente (nella quale si trova il programma in esecuzione).

Simile a **LD**…

***Istruzione STI (Store Indirect): scrittura indiretta in memoria***

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **STI** | **1** | **0** | **1** | **1** | **SR** | | | **pgoffset9** | | | | | | | | |

**SR → M(M(ind16))** **modalità di indirizzamento Indiretto**

Duale di LDI, l'istruzione porta il contenuto del registro sorgente SR nella cella il cui indirizzo è contenuto nella cella di memoria di indirizzo ind16. Serve dunque a scrivere il contenuto di una cella situata in qualsiasi posizione dello spazio di indirizzamento, ma richiede due accessi a memoria (uno per leggere l'indirizzo, l'altro per scrivere il dato).

Simile a **LDI**…

***Istruzione STR (Store using Register): scrittura in memoria con registro base***

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **STR** | **0** | **1** | **1** | **1** | **SR** | | | **BR** | | | **index6** | | | | | |

**SR → M(BR + $|index6_{16}|$)** **modalità di indirizzamento Base+offset**

Duale di STR, l'istruzione porta il contenuto del registro sorgente SR nella cella il cui indirizzo viene calcolato sommando al contenuto del registro base BR il valore a 16 bit ottenuto come estensione in valore assoluto della costante a 6 bit **index6** presente nell'istruzione:

$|index6_{16}|$ = 0000 0000 00 index6

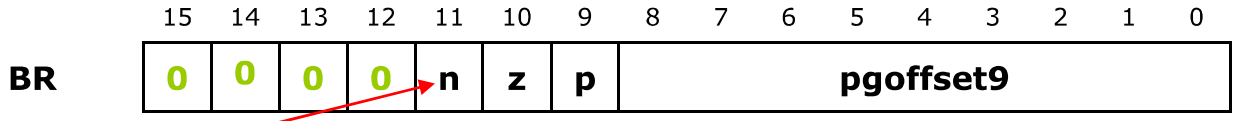
Serve dunque a scrivere il contenuto di una cella situata in qualsiasi posizione dello spazio di indirizzamento, con un solo accesso a memoria.

Simile a **LDR.**

### 1.2.3 Istruzioni di controllo

Anche per le istruzioni di controllo, il meccanismo per la costruzione dell'indirizzo a 16 bit mediante page offset è lo stesso descritto in precedenza.

***Istruzione BR (Branch): salto condizionato***




***if* (CC AND nzp ≠ 0) *then* ind16 → PC**

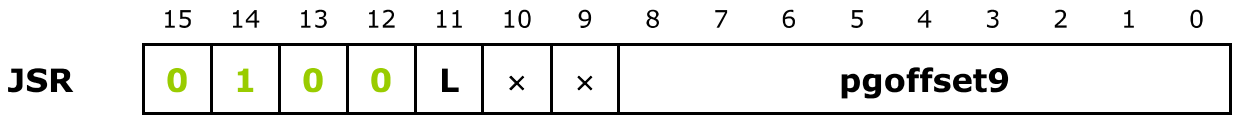
Se una delle condizioni specificate nell'istruzione (**n**, **z**, **p**) sono soddisfatte dal contenuto del registro ***CC*** (*Condition Codes*) viene seguito un salto condizionato all'istruzione contenuta nella cella di indirizzo **ind16.** La stessa istruzione può essere utilizzata per effettuare un salto incondizionato, attivando tutte le condizioni **n**, **z** e **p**.

Questa istruzione può essere chiamata in varie modalità a seconda che si voglia eseguire un salto incondizionato oppure un salto condizionato. Dopo un'istruzione di salto il Program Counter conterrà l'indirizzo contenuto nell'istruzione di salto: **ind16 → PC**. Questo è però vero in parte per i salti condizionati, nei quali **ind16 → PC** si verifica SOLO SE la condizione del salto è verificata.

Esempi di chiamata possono quindi essere:

```
BR x3001      ;la prossima istruzione da eseguire sarà quella con indirizzo x3001
              ;(salto INCONDIZIONATO)
BRP x3001     ;esegui il salto se CC = 001;
BRZ x3001     ;esegui il salto se CC = 010;
BRN x3001     ;esegui il salto se CC = 100;
BRZP x3001    ;esegui il salto se CC = 001 oppure CC=010;
BRNZP x3001   ;esegui il salto se CC = 001 oppure CC=010 oppure CC=100;
              ;cioè sempre (salto INCONDIZIONATO)
```

***Istruzione JSR (Jump to Subroutine): chiamata a sottoprogramma***




***if* (L = 1) *then* PC → R7 ; ind16 → PC**

L'istruzione effettua un salto incondizionato all'indirizzo **ind16**, ma preceduto dalla ricopiatura del valore corrente del PC in R7. Consente quindi di realizzare la chiamata a sottoprogramma: l'istruzione RET (descritta in seguito) consente di ritornare a eseguire l'istruzione immediatamente seguente la chiamata a sottoprogramma JSR.

Si noti che l'istruzione usa il registro R7 per salvare l'indirizzo di ritorno, quindi consente un solo livello di annidamento nelle chiamate a sottoprogramma (una chiamata successiva cancella l'indirizzo di ritorno della precedente).

Se il bit L vale 0, l'istruzione equivale a un semplice salto incondizionato.

Esempio:

**PC = x3001** ;PC prima del salto

**JSR x302A**

→ **R7 = x3001** ;salvo il PC prima del salto in R7. Il contenuto di R7 tornerà ;utile con una successiva operazione di **RET**.

→ **PC = x302A** ;salto incondizionato

***Istruzione JSRR (Jump to Subroutine using Register): chiamata a sottoprogramma con registro base***

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **JSRR** | 1 | 1 | 0 | 0 | L | × | × | BR | | | index6 | | | | | |

***if* (L = 1) *then* PC → R7 ; BR + $|index6_{16}|$ → PC**

Analoga a JSR, l'istruzione effettua un salto incondizionato all'indirizzo calcolato sommando al contenuto del registro base BR il valore a 16 bit ottenuto come estensione in valore assoluto della costante a 6 bit **index6** presente nell'istruzione, ma preceduto dalla ricopiatura del valore corrente del PC in R7. Consente quindi di realizzare la chiamata a sottoprogramma con modalità di indirizzamento base+offset (quindi dinamicamente variabile).

Come per JSR, anche l'istruzione JSRR usa il registro R7 per salvare l'indirizzo di ritorno, quindi consente un solo livello di annidamento nelle chiamate a sottoprogramma (una chiamata successiva cancella l'indirizzo di ritorno della precedente).

Se il bit L vale 0, l'istruzione equivale a un semplice salto incondizionato con indirizzamento base+offset.

Esempio:

**PC = x3001** ;PC prima del salto

**JSRR R0, #35** ;salto incondizionato a sottoprogramma

**R0 = x4123** ;R0 è un possibile registro base (in questo caso **BR**=**B**ase ;**R**egister)

**index6 = #35 → 100011 → $|index6_{16}|$ = 0000 0000 0010 0011**

**R7 = x3001** ;salvo il PC prima del salto in R7. Il contenuto di R7 tornerà ;utile con una successiva operazione di **RET**.

**R0 + $|index6_{16}|$ = x4146 → 0100 0001 0100 0110**

**PC = x4146** ;salto incondizionato

***Istruzione RET (Return): ritorno da sottoprogramma***

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **RET** | 1 | 1 | 0 | 1 | × | × | × | × | × | × | × | × | × | × | × | × |

**R7 → PC**

L'istruzione effettua il ritorno da sottoprogramma, poiché va a eseguire l'istruzione immediatamente seguente a quella che aveva effettuato la chiamata a sottoprogramma (JSR o JSRR).

Esempio:

**RET** ;PC = R7

***Istruzione RTI (Return from Interrupt): ritorno da interrupt***

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **RTI** | **1** | **0** | **0** | **1** | × | × | × | × | × | × | × | × | × | × | × | × |

**PCsaved → PC**

L'istruzione effettua il ritorno da interrupt.

L'interrupt e le relative modalità di funzionamento del calcolatore saranno affrontati nel Modulo 7.

***Istruzione TRAP: chiamata a sottoprogramma mediante vettore di trap***

| | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **TRAP** | **1** | **1** | **1** | **1** | × | × | × | × | **trapvect8** | | | | | | | |

**PC → R7 ; M(0000 0000 trapvect8) → PC**

L'istruzione effettua una chiamata a sottoprogramma all'indirizzo contenuto nella cella di indirizzo:

0000 0000 trapvect8.

A questo scopo, nello spazio di indirizzamento della CPU LC-2 vengono riservate le prime $2^8$=256 celle della memoria ad altrettanti indirizzi di sottoprogrammi, ciascuno associato a una posizione nel ***vettore di trap***.

Questa modalità di chiamata a sottoprogramma consente di disaccoppiare chiamante e chiamato: il chiamante deve solo sapere cosa fanno i 256 sottoprogrammi chiamabili, non dove sono situati, poiché basta la loro posizione del vettore di trap per raggiungerli. Viene usata tipicamente per le chiamate al Sistema Operativo, che offre servizi predefiniti rispetto al programma applicativo.

Sono previste tre semplici routine di interazione, associate ad altrettante posizioni nel **vettore di trap**:

**TRAP x21** ;emette su video il carattere il cui codice ASCII è contenuto in R0

**TRAP x22** ;emette su video la stringa di caratteri che inizia nella cella
;il cui indirizzo è contenuto in R0 e che termina con il carattere NUL (x00)

**TRAP x23** ;legge un carattere da tastiera e ne riporta il codice ASCII in R0

**TRAP x25** ;arresta l'esecuzione del programma