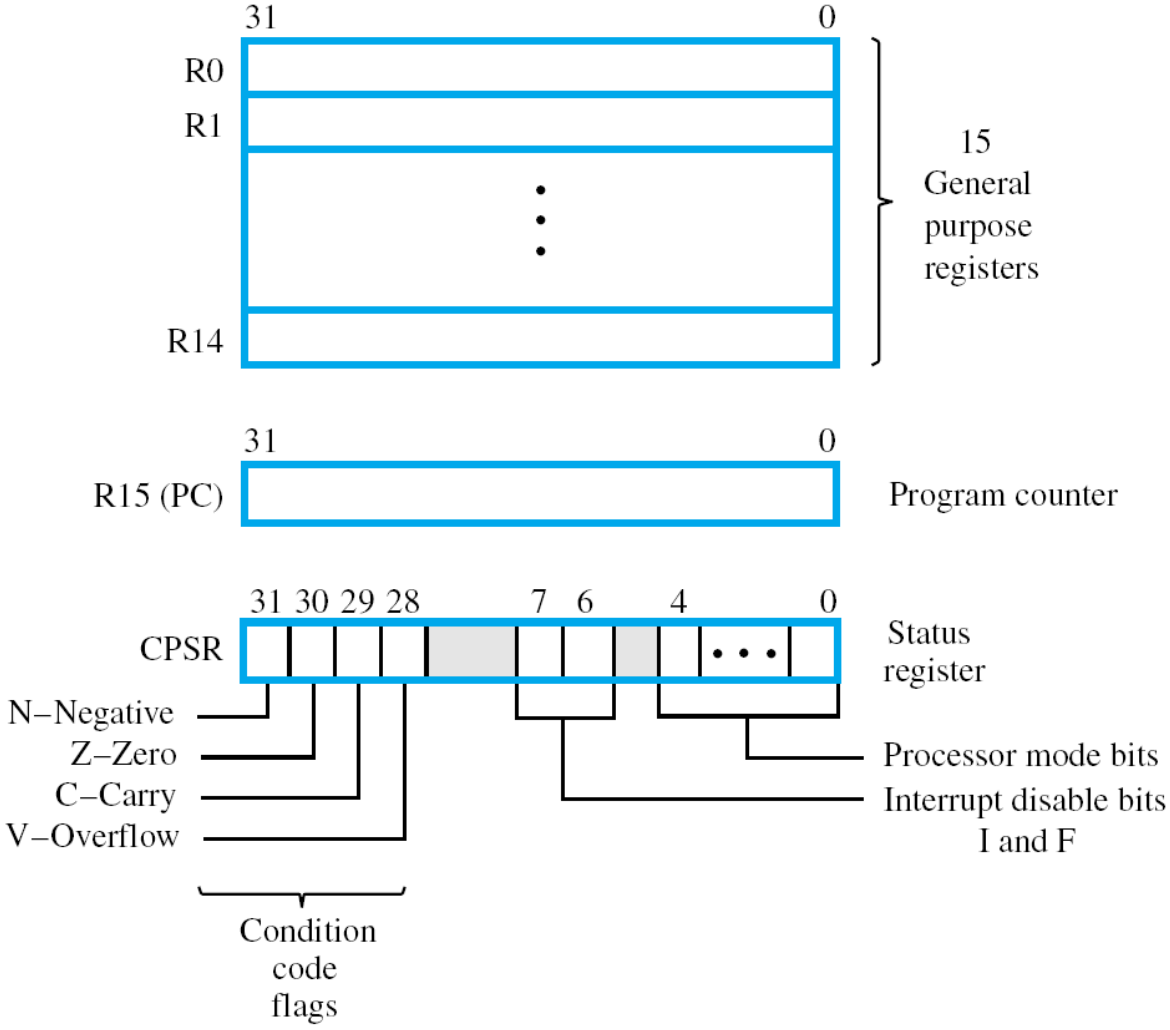
Arm ISA

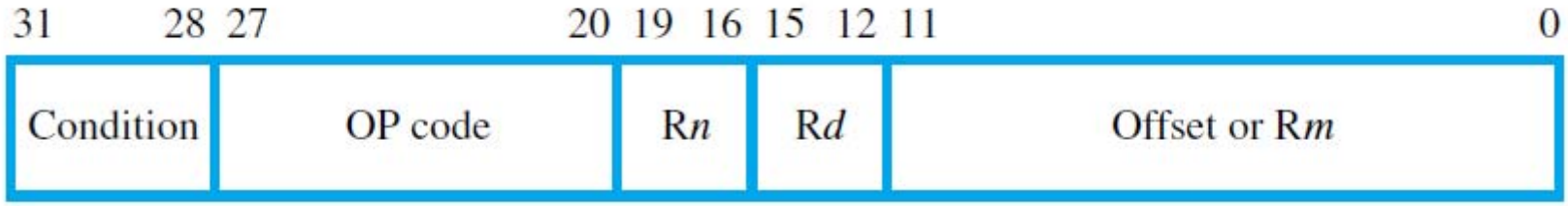
Registers

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

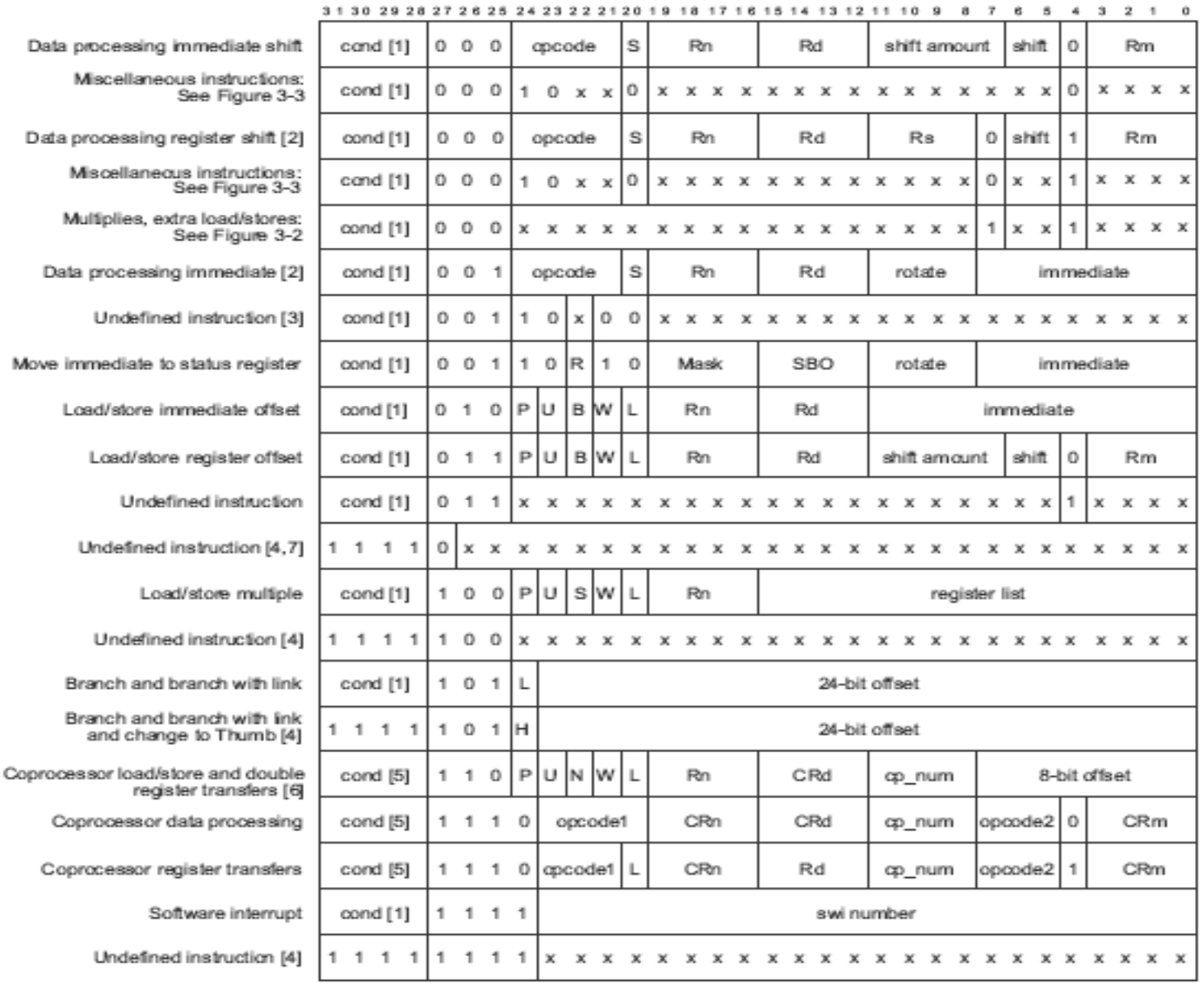
Status Register



Codifica dell'istruzione LOAD/STORE (LDR/STR)



Codifica istruzioni ARM



Modi di Indirizzamento

- Tutte le istruzioni fanno riferimento a registri o immediati, tranne LDR e STR
- Modi di indirizzamento per gli operandi in memoria
  - Tutti derivati dal indexed addressing
  - L' effective address dell'operando è espresso come somma di un registro di base (R*n*) e un offset con segno
  - L'offset può essere un immediato a 12-bit o il contenuto di un secondo registro R*m*
  - Nelle prossime slide sono riportati degli esempi per LDR ed STR
    - Entrambe le operazioni accedono ad una word

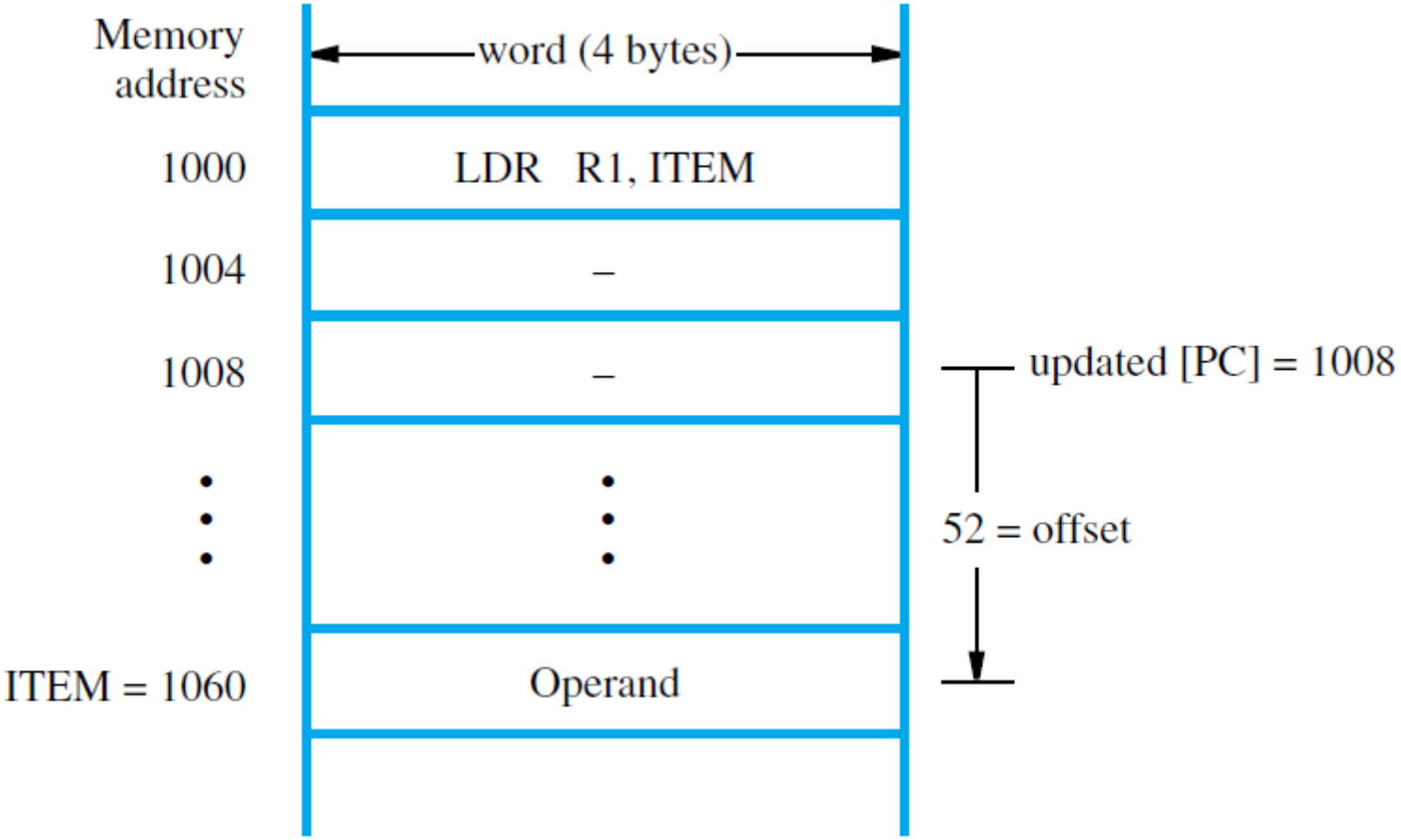
Addressing modes

- Relative mode
  - Sostituisce il modo **ASSOLUTO** poiché una istruzione da 32 bit non può contenere un indirizzo di 32 bit
  - L'indirizzo destinazione è dunque ottenuto come spiazzamento rispetto al PC, lo spiazzamento è calcolato dall'assemblatore

```
LDR  Rd, ITEM
```

in realtà è eseguito come

$$Rd \leftarrow [[PC] + offset]$$


Memory address
word (4 bytes)
1000
LDR R1, ITEM
1004
–
1008
–
updated [PC] = 1008
52 = offset
ITEM = 1060
Operand

Pre-Indexed

- Pre-indexed mode:

  `LDR  Rd, [Rn, #offset]` $Rd \leftarrow [[Rn] + offset]$

- Pre-indexed with writeback:

  `LDR  Rd, [Rn, #offset]!` $Rd \leftarrow [[Rn] + offset]$, $Rn \leftarrow [Rn] + offset$

  - Rm può sostituire #offset

    `LDR  Rd, [Rn, Rm]` $Rd \leftarrow [[Rn] + [Rm]]$

    - In questo caso il contenuto di Rm può essere shiftato prima dell'uso (vedi prox slide)

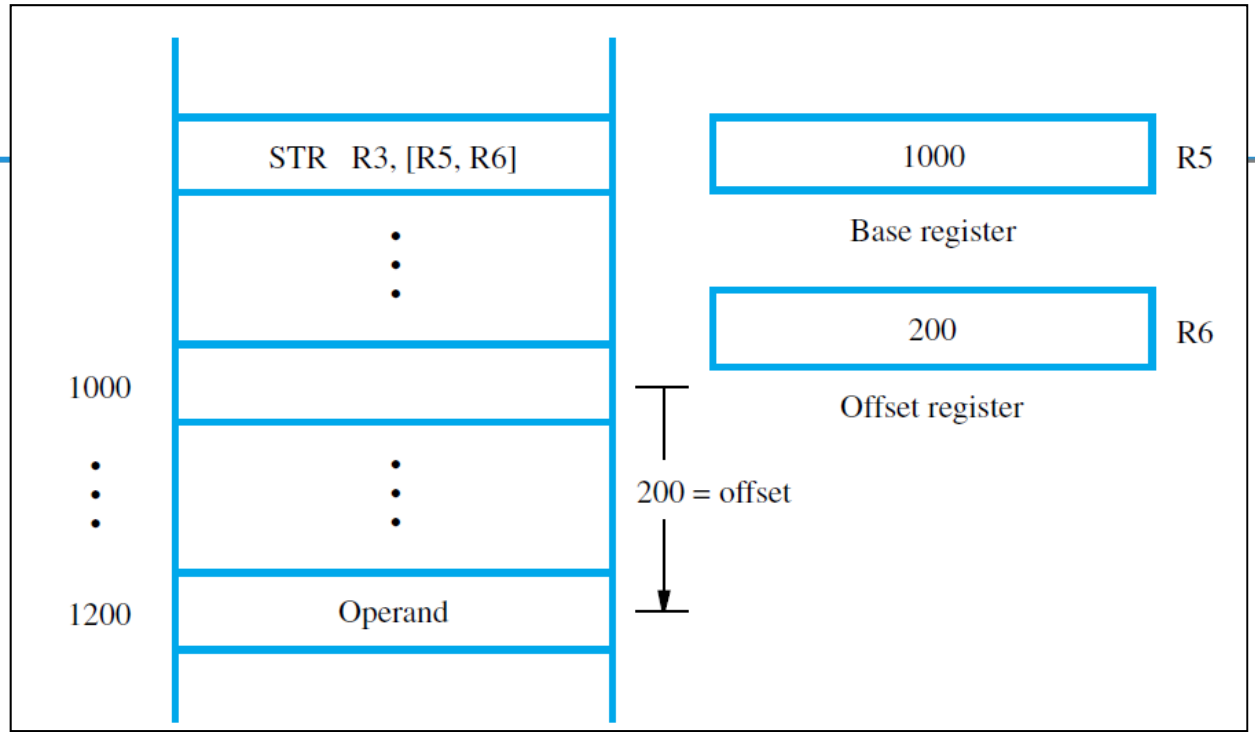
Shift dell'operando

```
LDR     R0,[R1, –R2, LSL #4]!
```

R0←[[R1]–16×[R2]]
R1←[R1]–16×[R2]

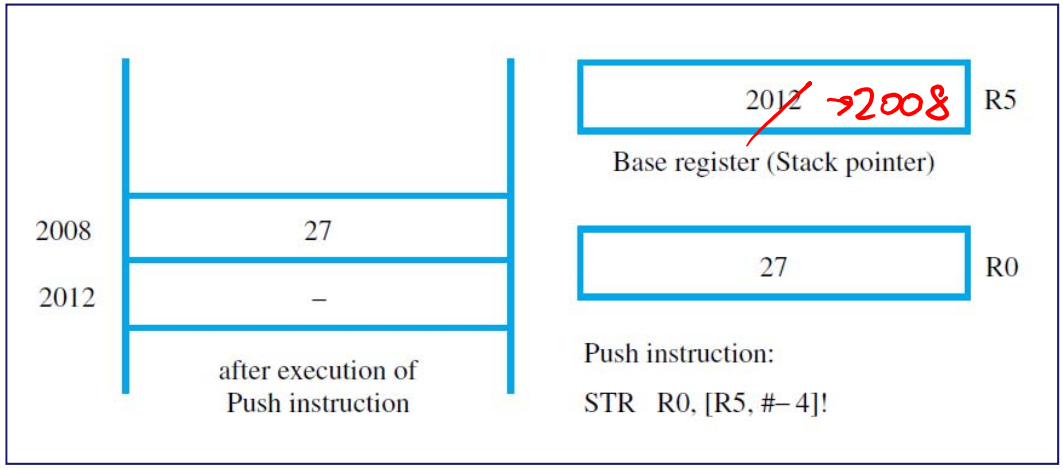
- La rotazione può essere:

  Logica, sinistra e destra LSL/LSR



Pre-indexed

Pre-indexed
With writeback



Post-indexed

`LDR  R`*d*`, [R`*n*`], #offset`

R*d* ← [[R*n*]]
R*n* ← [R*n*] + offset

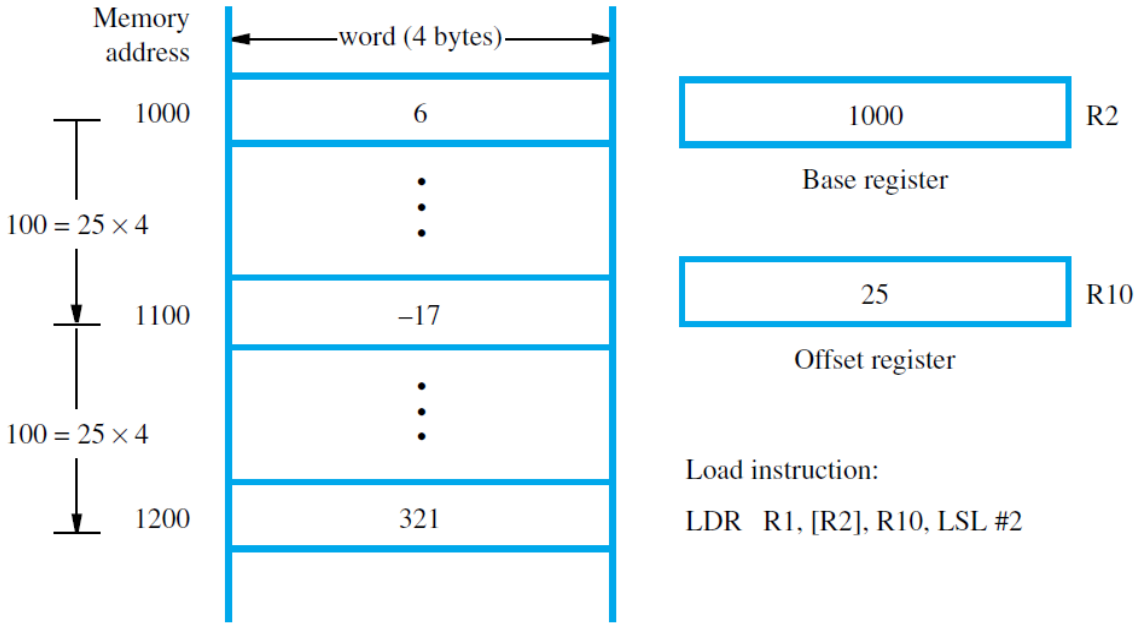
- R*m* può sostituire #offset
  - In questo caso il contenuto di R*m* può essere shiftato prima di essere utilizzato

`LDR  R0,[R1], R10, LSL #2`

R0←[[R1]]
R1←[R1]+4×[R10]

Pre-Index Vs Post-Index

```
PRE:     LDR     R1,[R2, R10, LSL #2]
PRE/WB:  LDR     R1,[R2, R10, LSL #2]!
POST:    LDR     R1,[R2], R10, LSL #2
```



Esecuzione 1	Esecuzione 2
Caso PRE: R1 = -17 R2 = 1000 Caso PRE/WB: R1 = -17 R2=1100 Caso POST: R1=6 R2=1100	Caso PRE: R1 = -17 R2 = 1000 Caso PRE/WB: R1 = 321 R2=1200 Caso POST: R1=-17 R2=1200

ARM indexed addressing modes.

Name	Assembler syntax	Addressing function
With immediate offset:		
Pre-indexed	[R*n*, #offset]	EA = [R*n*] + offset
Pre-indexed with writeback	[R*n*, #offset]!	EA = [R*n*] + offset; R*n* ← [R*n*] + offset
Post-indexed	[R*n*], #offset	EA = [R*n*]; R*n* ← [R*n*] + offset
With offset magnitude in R*m*:		
Pre-indexed	[R*n*, ± R*m*, shift]	EA = [R*n*] ± [R*m*] shifted
Pre-indexed with writeback	[R*n*, ± R*m*, shift]!	EA = [R*n*] ± [R*m*] shifted; R*n* ← [R*n*] ± [R*m*] shifted
Post-indexed	[R*n*], ± R*m*, shift	EA = [R*n*]; R*n* ← [R*n*] ± [R*m*] shifted
Relative (Pre-indexed with immediate offset)	Location	EA = Location = [PC] + offset

EA = effective address
offset = a signed number contained in the instruction
shift = direction #integer
where direction is LSL for left shift or LSR for right shift; and
integer is a 5-bit unsigned number specifying the shift amount
±R*m* = the offset magnitude in register R*m* can be added to or subtracted from the contents of base register R*n*

ARM

Le principali istruzioni

Direttive di AREA

- **AREA:** Informa l'assemblatore dell'inizio di una nuova area di **codice** o di **dati**

```
        AREA Example,CODE,READONLY      ; Definisce una nuova area di codice READONLY di
                                        ; nome Example
; codice
```

  - **GNU Assembly (quello che utilizzeremo)**

```
        AREA s, CODE        =>      .text
        AREA s, DATA        =>      .data
```

- **END:** informa l'assemblatore della fine del programma
  - GNU Assembly

```
        END                 =>      .end
```

Direttive per la definizione delle variabili

- ARM
  - **DCB**: riserva un'area di uno o più byte
  - **DCW**: riserva un'area di una o più halfword
  - **DCD**: riserva un'area di una o più word
  - **SPACE**: riserva un'area di memoria di n byte senza inizializzarla
  - Per le stringhe
    DCB "aaa"
  - **DCB** "aaa",0
  - **ALIGN** to realign the memory

- GNU
  - .byte

```
a: .byte 3,4,5
```

  - .word

```
var: .word 3
var: .word 3, 4, 5
```

  - .space *n*

```
var: .space 3
```

  - .ascii, .asciz/.string

```
mytxt .ascii  "aaa"
mytxt .asciz  "aaa"
mytxt .string "aaa"
```

  - .align

Template di programmma ARM

```
.text

; inserire il codice del programma
; che usa A, N, S1 ed S2

.data
A  .word 10, 7, 1, 3
N  .byte 4
S1 .string “questa è una stringa”
S2 .asciz  “anche questa è una stringa”

.end
```

Istruzioni di accesso alla memoria

- Load e Store:
  LDR / STR trasferiscono una parola dalla memoria/in memoria
  - Possono essere seguite dalla dimensione del dato (se diverso dalla parola)
    LDRH / STRH per half words (zero-extended in caso di Load)
    LDRB / STRB for bytes (zero-extended in caso di Load)
    LDRSH and LDRSB se necessario **sign-extended** Load

- Esempi:

```
LDR r9, =S          ; Carica l'indirizzo di N in r9
LDRB r8, [r9]       ; Carica un byte (carattere) della
                    ; stringa S
LDRB r8, [r9,#1]!    ; Carica il prossimo byte di S e
                     ; incrementa r9
LDRB r8, [r9],#1     ; Carica un byte da S e dopo
                     ;incrementa r9
STR r2, [r9]        ; memorizza una word in S
```

Load/Store Multipli

- Multiple-word Load / Store:
  LDM/STM seguito da IA IB DA DB:
  - I = Increment; D = Decrement
  - A = After; B = Before
- Esempio:

```
LDMIA  R10!, [R0, R1, R6, R7]
```

Se [R10] = 1000, le words a 1000, 1004, 1008, and 1012 sono caricate nei registry R0, R1, R6, R7, e al termine R10 aggiornato a 1016

Istruzioni Aritmetiche (1/2)

- Formato

  `OP  Rd, Rn, Rm` (**Rm** può essere sostituito da **#n** con n < 255)

- Esempi

```
ADD  R0, R2, R4      R0 ← [R2] + [R4]
SUB  R0, R3, #17     R0 ← [R3] – 17
MUL  R0, R1, R2      R0 ← [R1] × [R2]
```

- Le operazioni aritmetiche non alterano i bit di stato almeno non specificato con il suffisso **S**

```
ADDS R0, R2, R4      R0 ← [R2] + [R4] e setta codici condizione
```

Istruzioni Aritmentiche (2/2)

- Il secondo operando sorgente può essere ruotato prima di essere utilizzato

  `ADD R0, R1, R5, LSL #4` $R0 \leftarrow [R1] + 16 \times [R5]$

  - La rotazione può essere:

    Logica, sinistra e destra LSL/LSR

    Aritmetica, sinistra e destra ASR/ROR

- Per le operazioni vettoriali può essere utile ***Multiply and Accumulate***

  `MLA R0, R4, R5, R6` $R0 \leftarrow ([R4] \times [R5]) + [R6]$

  - Con R6=R0 può semplificare il prodotto vettoriale

- **Somma con riporto** (come full-Adder).
  Es. Somma su 64 bit R3:R2 e R5:R4 risultato in R7:R6

```
ADDS R6, R2, R4
ADC  R7, R3, R5
```

Istruzioni MOV

- Nella forma base

  `MOV  Rd, Rm` R*d* ← [R*m*]

  Con Rm che può essere sostituito da un immediato #*n*

- Il secondo operando può essere shiftato/ruotato

  `MOV  Ri, Rj, LSL #4` R*i* ← [R*j*] x 16

  equivale a

  `LShiftL  Ri, Rj, #4`

- Usando **MVN** in luogo di MOV il secondo operando è complementato bit a bit prima di essere copiato

Istruzioni Logiche

- Operano un confront bit a bit degli operandi sorgente e memorizzano il risultato nel registro destinazione

```
AND   Rd, Rn, Rm   (bit-wise logical AND)
ORR   Rd, Rn, Rm   (bit-wise logical OR)
EOR    Rd, Rn, Rm  (bit-wise logical XOR)
```

  - Non settano codici di condizione se non è specificato il suffisso **S**
- `BIC  Rd, Rn, Rm` copia R*n* in R*d* azzerando i bit alti in Rm
  - Equivale a AND tra R*n* e il complemento di R*m*

```
BIC      R0, R0, R1   R0=02FA62CA R1= 0000FFFF =>
                      R0= 02FA0000
```

Istruzioni di Test

- A differenza delle istruzioni logiche settano sempre i codici di condizione:
  - TST R*n*, R*m* (o #*n*) Effettua una AND e setta i codici di condizione

    ESEMPIO:

    `TST R3, #1` Setta Z = 1 se il bit basso di R3 è 0

    Setta Z = 0 se il bit basso di R3 è 1
  - TEQ Rn, Rm (o #n) Effeffua la XOR dei valori e setta Z (Z=1 se i valori sono uguali)

    ESEMPIO:

    `TEQ R2, #5` Z=1 se R2 contiene 5
  - CMP Rn, Rm Calcola Rn-Rm e setta i codici di cond.

Esempio di programma

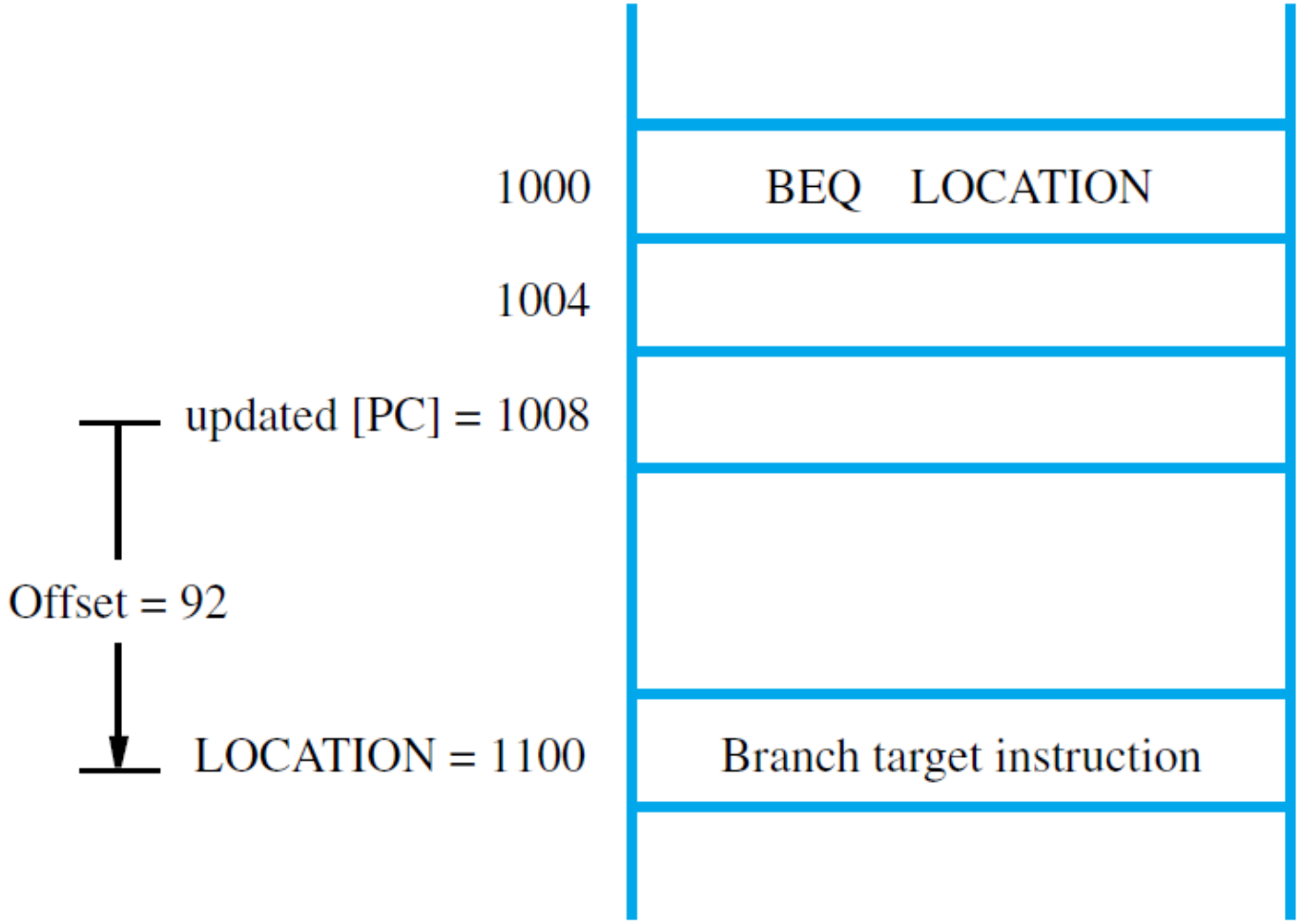
```
.text
main:
; Inizializzazione dei registri
        LDR  r0,    =vet1
        LDR r1,     =vet2
        LDR r2,     =prodVett
        MOV r3,   #0
; Prodotto tra vettori
        LDRB r4,[r0], #1
        LDRB r5,[r1], #1
        MLA r3, r4, r5, r3      ; prodotto 1^ comp.

        LDRB r4,[r0], #1
        LDRB r5,[r1], #1
        MLA r3, r4, r5, r3      ; prodotto 2^ comp.
        LDRB r4,[r0], #1
        LDRB r5,[r1], #1
        MLA r3, r4, r5, r3      ; prodotto 3^ comp.

; Memorizza risultato
        STR r3, [r2]

fine:
   swi 0x11

.data
vet1:       .byte 3, 5, 6
.align
vet2:       .byte 1, 4, 7
.align
prodVett: .skip 1

.end
```

Istruzioni di Controllo Flusso

- BRANCH: sono sempre condizionate dal valore dei codici di condizione

  B***CC*** DESTINAZIONE

  - ***CC*** rappresenta la condizione da verificare
    Esempio:
    B*EQ* LOCATION Salta a LOCATION se Z = 1


1000
BEQ LOCATION
1004
updated [PC] = 1008
Offset = 92
LOCATION = 1100
Branch target instruction

Condition field encoding in ARM instructions.

Condition field $b_{31} \ldots b_{28}$	**Condition suffix**	**Name**	**Condition code test**
0 0 0 0	EQ	Equal (zero)	Z = 1
0 0 0 1	NE	Not equal (nonzero)	Z = 0
0 0 1 0	CS/HS	Carry set/Unsigned higher or same	C = 1
0 0 1 1	CC/LO	Carry clear/Unsigned lower	C = 0
0 1 0 0	MI	Minus (negative)	N = 1
0 1 0 1	PL	Plus (positive or zero)	N = 0
0 1 1 0	VS	Overflow	V = 1
0 1 1 1	VC	No overflow	V = 0
1 0 0 0	HI	Unsigned higher	$\overline{C} \vee Z = 0$
1 0 0 1	LS	Unsigned lower or same	$\overline{C} \vee Z = 1$
1 0 1 0	GE	Signed greater than or equal	$N \oplus V = 0$
1 0 1 1	LT	Signed less than	$N \oplus V = 1$
1 1 0 0	GT	Signed greater than	$Z \vee (N \oplus V) = 0$
1 1 0 1	LE	Signed less than or equal	$Z \vee (N \oplus V) = 1$
1 1 1 0	AL	Always	
1 1 1 1		not used	

Strutture di controllo elementare: IF…THEN…

```
if condizione {
   // codice vera
}
// codice
```

Esempio

```
if isEven(a){
   pari = 1;
}
// caso dispari
```

```
;   Check !condizione
    BEQ non_vera
;   codice vera
non_vera:
;       codice
```

Esempio

```
    LDR R3, =a
    LDR R3, [R3]
    LDR R2, =pari
    TST R3, #1
    BNE dispari
    MOV R1, #1
    STR R1, [R2]
dispari:
; caso dispari
    MOV R1, #2
    STR R1, [R2]
```

Strutture di controllo elementare: IF…THEN…ELSE

```
if condizione {
     // codice vero
} else {
     // codice falso
}
// codice
```

Esempio

```
if isEven(a){
     pari = 1;
} else
     pari = 0;
}
// codice
```

```
     Check cond.
     B(condizione falsa) falso
;         codice vero
     B fine
falso:
;         codice falso
Fine:
;    codice
```

Esempio

```
     LDR R3, =a
     LDR R3, [R3]
     LDR R2, =pari
     TST  R3, #1
     BNE  dispari
     MOV R1, #1
     STR R1, [R2]
     B fine
dispari:
; caso dispari
     MOV R1, #0
     STR R1, [R2]
fine:
```

Strutture di controllo elementare: FOR …

```
for (i = 0; i <= N; i ++){
// codice
}
// fuori for
```

```
        MOV R1, #0

        BAL for
next:ADD R1,R1,#1

;  codice

for: CMP R1,#N
     BLT next
; fuori for
```

Strutture di controllo elementare: WHILE …

```
while (condizione){
// codice
}
// fuori while
```

```
   BAL check
loop:
;codice
check:
   B{condizione} loop

; fuori while
```

Somma gli elementi di un vettore NUM1

```
1  int NUM1 = {1,2,3,4};
2  int N = 4;
3  int SUM;
4
5  r1 = N;
6  r2 = NUM1;
7  r0 = 0;
8
9  do{
10      r3 = *(r2);
11      r2 = r2+4;
12      r0 = r0+r3;
13      r1 = r1 - 1;
14 }while(r1 > 0)
15 SUM = r0
```

```
        LDR   r4, =SUM

        LDR   r1, N
        LDR   r2, =NUM1
        MOV   r0, #0

LOOP:
        LDR   r3, [r2], #4

        ADD   r0, r0, r3
        SUBS  r1, r1, #1
        BGT   LOOP
        STR   r0, [r4]
```

Esempio di programma ARM

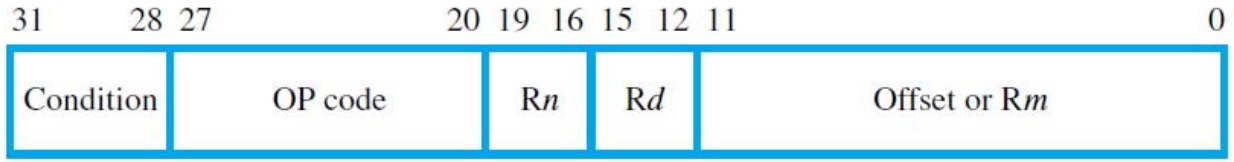
Si consideri il seguente codice
(Algoritmo di Euclide per il Massimo Comune Divisore):

```
function gcd (integer a, integer b)
: result is integer
    while (a<>b) do
     if (a > b) then
          a = a - b
     else
          b = b - a
     endif
    endwhile
    result = a
```

```
gcd
        CMP     r0,r1
        BEQ     end
        BLT     less
        SUB     r0,r0,r1
        BAL     gcd
less
        SUB     r1,r1,r0
        BAL     gcd
end
```

Conditional Execution

- Quasi tutte le istruzioni ARM possono essere condizionate allo stato del processore (Codici condizione)
- I bit 28-31 dell'istruzione riportano la condizione da verificare affinché l'istruzione venga eseguita



- Esempi:

```
ADD r0, r1, r2      ; r0 = r1 + r2, non aggiorna codici condizione
ADDS r0, r1, r2     ; r0 = r1 + r2, e aggiorna codici condizione
ADDSCS r0, r1, r2   ; se C flag set allora r0 = r1 + r2, e aggiorna
                    ; codici condizione
```

Esempio di uso dei codici di condizione

Consideriamo lo stesso esempio di prima

```
gcd
        CMP    r0,r1
        BEQ    end
        BLT    less
        SUB    r0,r0,r1
        BAL    gcd
less
        SUB    r1,r1,r0
        BAL    gcd
end
```

```
gcd
        CMP r0,r1
        SUBGT r0,r0,r1
        SUBLT r1,r1,r0
        BNE gcd
end
```

Conditional Vs Non Conditional

r0: a	r1: b	Instruction	Cycles (ARM7)
1	2	CMP r0, r1	1
1	2	BEQ end	1 (not executed)
1	2	BLT less	3
1	2	SUB r1, r1, r0	1
1	2	B gcd	3
1	1	CMP r0, r1	1
1	1	BEQ end	3
			Total = 13

r0: a	r1: b	Instruction	Cycles (ARM7)
1	2	CMP r0, r1	1
1	2	SUBGT r0,r0,r1	1 (not executed)
1	1	SUBLT r1,r1,r0	1
1	1	BNE gcd	3
1	1	CMP r0,r1	1
1	1	SUBGT r0,r0,r1	1 (not executed)
1	1	SUBLT r1,r1,r0	1 (not executed)
1	1	BNE gcd	1 (not executed)
			Total = 10

case where r0 equals 1 and r1 equals 2

Condition Code Suffixes

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned >=)
CC or LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always. This suffix is normally omitted.