Architettura degli Elaboratori I - B

Formato di Istruzione

Data Processing

Daniel Riccio/Alberto Aloisio

Università di Napoli, Federico II

27 marzo 2018

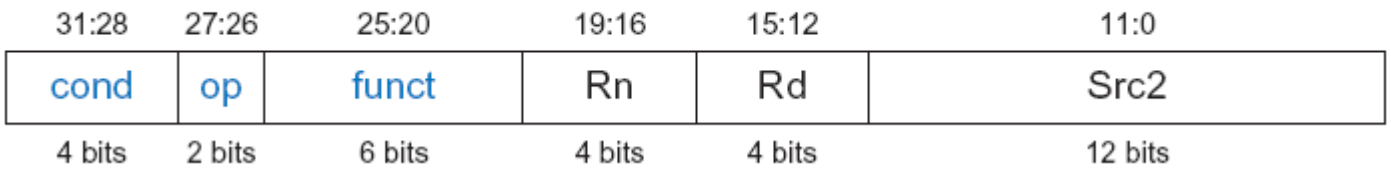
Rif. Capitolo 6

Digital Design and Computer Architecture-ARM, Harris-Harris, Edition-Morgan Kaufmann.

Istruzioni di Data-Processing

Il formato delle istruzioni di **data-processing** è il più comune.
Il primo operando sorgente è un registro. Il secondo operando sorgente può essere una costante o un registro. La destinazione è un registro.

Data-processing



La funzione eseguita è codificata nei campi evidenziati in blu:
op – chiamato anche codice codice operativo (è impostato a 2 per le istruzioni di data-processing)
funct – è chiamato anche function code;
cond – regola le condizioni di esecuzione sulla base di flag.

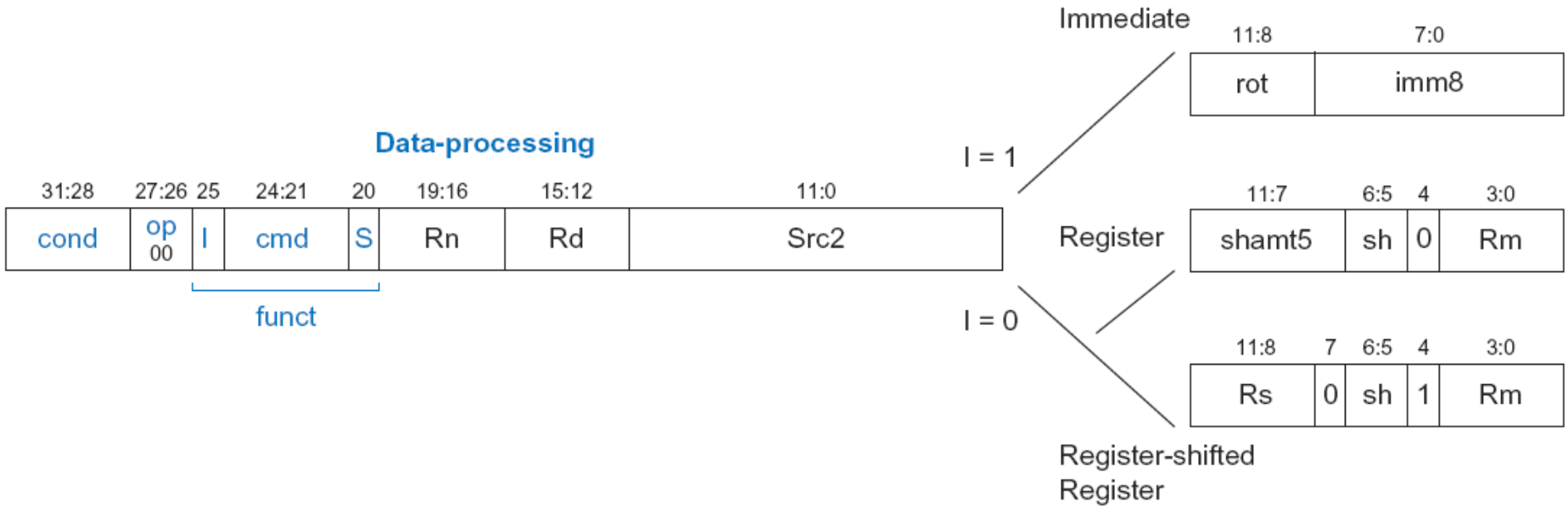
Gli operandi sono codificati nei tre campi **Rn**, **Rd**, e **SRC2**.
Rn è il primo registro sorgente, **SRC2** è la seconda sorgente e **Rd** indica il registro destinazione.

Istruzioni di Data-Processing

Il campo **funct** ha tre sottocampi:

- **I** – è 1 quando **Src2** è una costante;
- **cmd** – indica il tipo di istruzione;
- **S** – è 1 quando l'istruzione imposta i flag di condizione.

I bit **Src2** possono rappresentare una costante o un registro, il cui valore può essere opzionalmente shiftato rispetto ad una costante o al valore contenuto in un altro registro.



Traduzione di istruzioni

ADD R5, R6, R7;

Istruzione

ADD
- **op** = 00
- **funct** = 0100

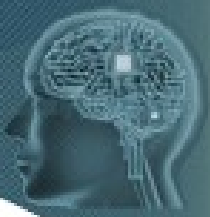
Secondo operando registro
Nessuna condizione
- **I** = 0
- **S** = 0

Registri
- **R5**
- **R6**
- **R7**

Cond	op	I	funct	S	R6	R5	Shamt5	Sh		R7
1110	00	0	0100	0	6	5	0	0	0	7

1110	00	0	0100	0	0110	0101	0	0	0	0111

Istruzioni di Data-Processing



Data-processing instructions with three register operands

Assembly Code

```
ADD R5, R6, R7
    (0xE0865007)
SUB R8, R9, R10
    (0xE049800A)
```

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110_2	00_2	0	0100_2	0	6	5	0	0	0	7
1110_2	00_2	0	0010_2	0	9	8	0	0	0	10
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
1110	00	0	0100	0	0110	0101	00000	00	0	0111
1110	00	0	0010	0	1001	1000	00000	00	0	1010
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Data-processing instructions with an immediate and two register operands

Assembly Code

```
ADD R0, R1, #42
    (0xE281002A)
SUB R2, R3, #0xFF0
    (0xE2432EFF)
```

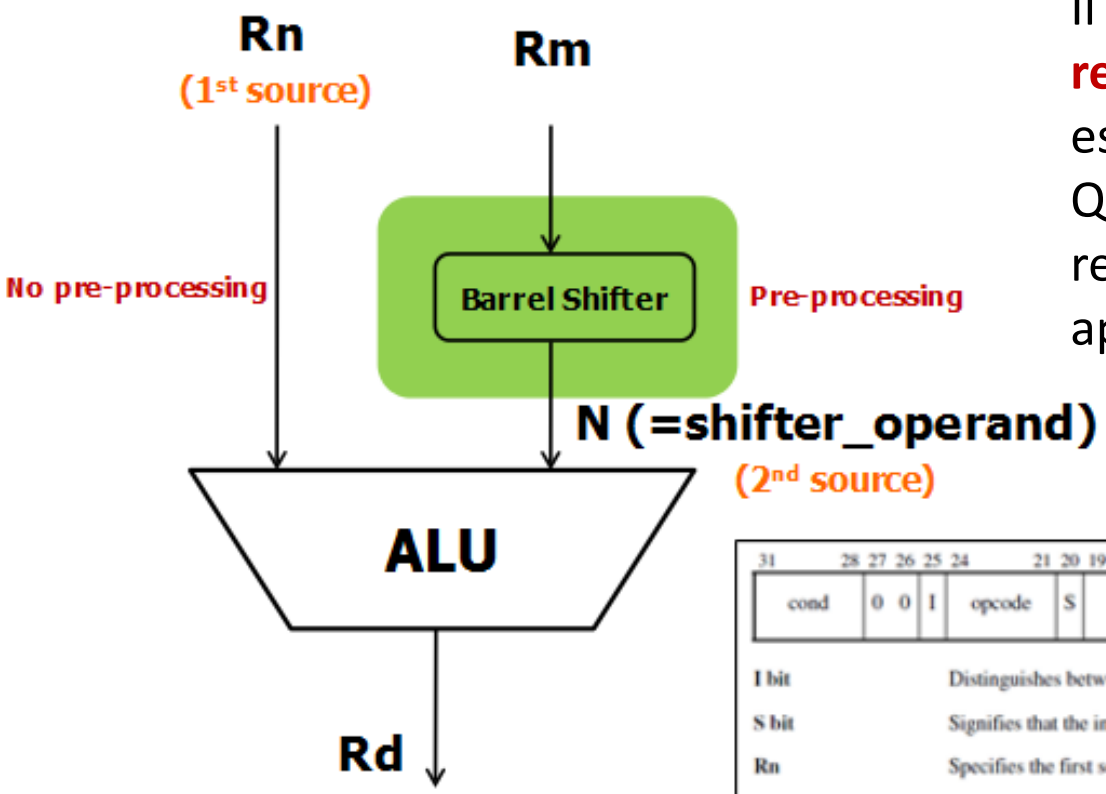
Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0100_2	0	1	0	0	42
1110_2	00_2	1	0010_2	0	3	2	14	255
cond	op	I	cmd	S	Rn	Rd	rot	imm8

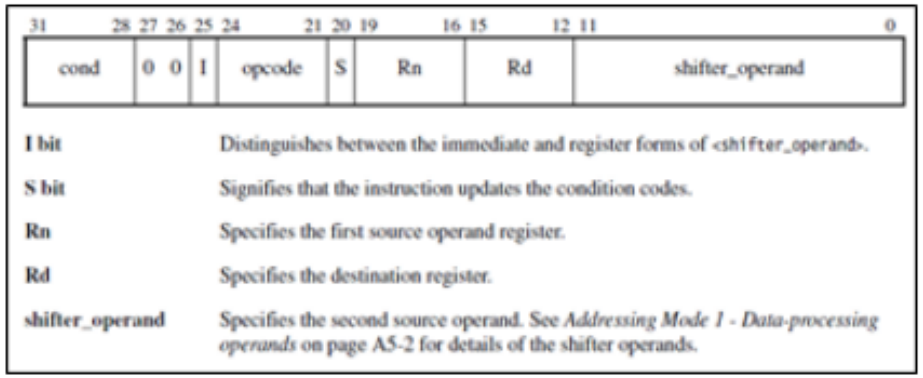
Machine Code

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110	00	1	0100	0	0001	0000	0000	00101010
1110	00	1	0010	0	0011	0010	1110	11111111
cond	op	I	cmd	S	Rn	Rd	rot	imm8

Istruzioni di data processing



Il primo input è sempre un **registro**, mentre il secondo può essere un **registro** o una **costante**. Quando il secondo operando è un registro, su di esso può essere applicato uno **shift**.



I bit	Distinguishes between the immediate and register forms of <shifter_operand>.
S bit	Signifies that the instruction updates the condition codes.
Rn	Specifies the first source operand register.
Rd	Specifies the destination register.
shifter_operand	Specifies the second source operand. See *Addressing Mode 1 - Data-processing operands* on page A5-2 for details of the shifter operands.

Istruzioni di data processing

L'architettura ARM definisce una serie di istruzioni di elaborazione dati, spesso chiamate istruzioni logiche e aritmetiche.

Istruzioni logiche e aritmetiche

Le operazioni logiche in ARM includono ORR (OR), EOR (XOR) e BIC (bit clear). Esse operano bit a bit su due input e scrivono il risultato in un registro di destinazione.

Il primo input è sempre un registro, mentre il secondo può essere un registro o una costante. Un'altra operazione logica, MVN (MoVe and Not), esegue un NOT bit a bit sul secondo input e scrive il risultato in un registro di destinazione.

Assembly code

```
AND  R3, R1, R2
ORR  R4, R1, R2
EOR  R5, R1, R2
BIC  R6, R1, R2
MVN  R7, R2
```

Source registers

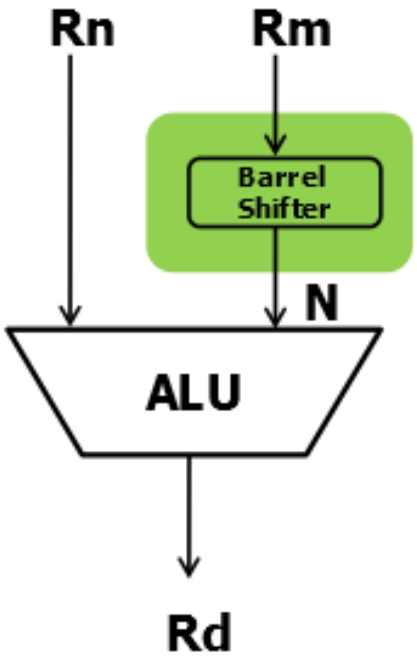
R1	0100 0110	1010 0001	1111 0001	1011 0111
R2	1111 1111	1111 1111	0000 0000	0000 0000

Result

R3	0100 0110	1010 0001	0000 0000	0000 0000
R4	1111 1111	1111 1111	1111 0001	1011 0111
R5	1011 1001	0101 1110	1111 0001	1011 0111
R6	0000 0000	0000 0000	1111 0001	1011 0111
R7	0000 0000	0000 0000	1111 1111	1111 1111

Istruzioni di data processing (logiche)

Nel linguaggio assembly del processore ARM, le istruzioni logiche hanno la seguente sintassi.

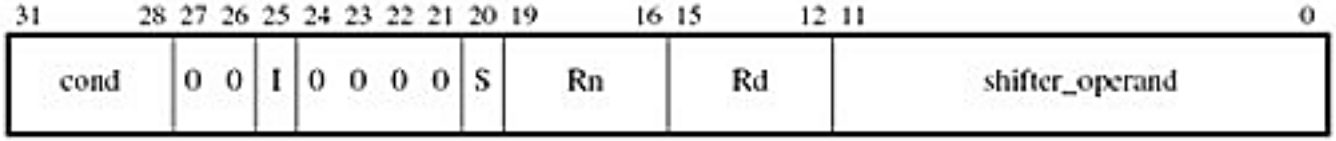


Syntax: <instruction>{cond}{S} Rd, Rn, N

AND	logical bitwise AND of two 32-bit values	*Rd = Rn & N*
ORR	logical bitwise OR of two 32-bit values	*Rd = Rn | N*
EOR	logical exclusive OR of two 32-bit values	*Rd = Rn ^ N*
BIC	logical bit clear	*Rd = Rn & ~N*

Istruzioni di data processing (AND)

Istruzione: **AND**



AND performs a bitwise AND of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the AND operation.

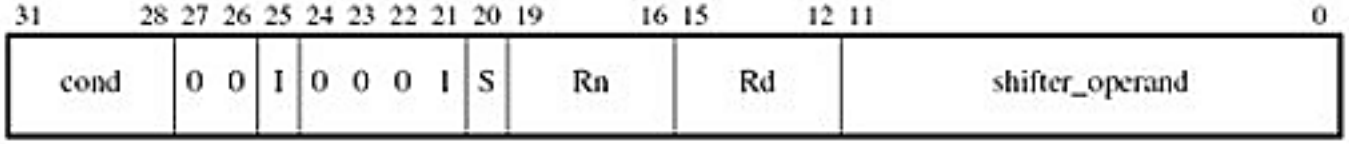
AND can optionally update the condition code flags, based on the result.

L'istruzione **AND** effettua l'and logico fra i due operandi e pone il risultato nel registro **Rd**.

```
AND  R0, R0, #3 ; Keep bits zero and one of R0 and discard the rest
```

Istruzioni di data processing (AND)

Istruzione: **EOR**



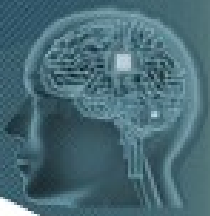
EOR (Exclusive OR) performs a bitwise Exclusive-OR of two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the exclusive OR operation.

EOR can optionally update the condition code flags, based on the result.

L'istruzione **EOR** effettua l'or esclusivo fra i due operandi e pone il risultato nel registro **Rd**.

```
EOR  R0, R0, #3 ; Invert bits zero and one of R0
```

Istruzioni di data processing (Esempi)



Before: r0 = 0x0000_0000
r1 = 0x0204_0608
r2 = 0x1030_5070

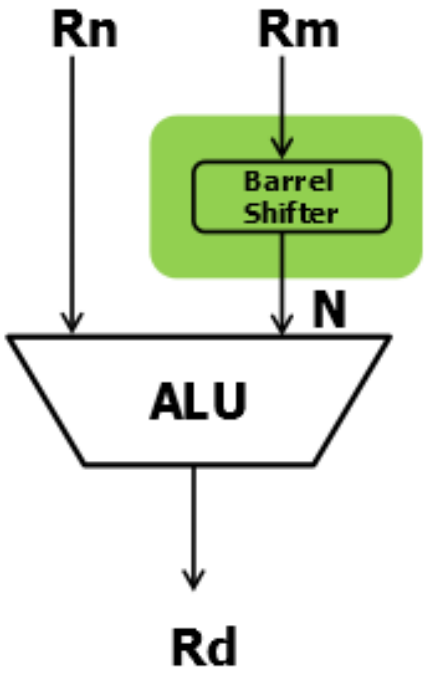
ORR r0, r1, r2

After:

r0 = 0x1234_5678

Istruzioni di data processing

Nel linguaggio assembly del processore ARM, le istruzioni aritmetiche hanno la seguente sintassi.

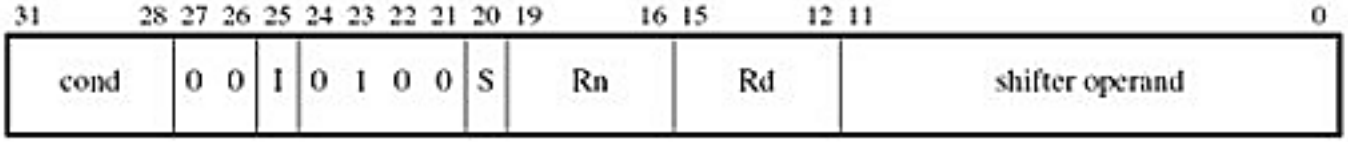


Syntax: <instruction>{cond}{S} Rd, Rn, N

ADC	add two 32-bit values with carry	*Rd = Rn + N + carry*
ADD	add two 32-bit values	*Rd = Rn + N*
RSB	reverse subtract of two 32-bit values	*Rd = N - Rn*
RSC	reverse subtract of two 32-bit values with carry	*Rd = N – Rn - !C*
SBC	subtract two 32-bit values with carry	*Rd = Rn - N - !C*
SUB	subtract two 32-bit values	*Rd = Rn - N*

Istruzioni di data processing (ADD)

Istruzione: **ADD**



ADD adds two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the addition.

ADD can optionally update the condition code flags, based on the result.

L'istruzione **ADD** somma due operandi e pone il risultato nel registro **Rd**.

```
ADD  R0, R1, R2 ; R0 = R1 + R2
ADD  R0, R1, #256 ; R0 = R1 + 256
ADDS R0, R2, R3,LSL#1 ; R0 = R2 + (R3 << 1) and update flags
```

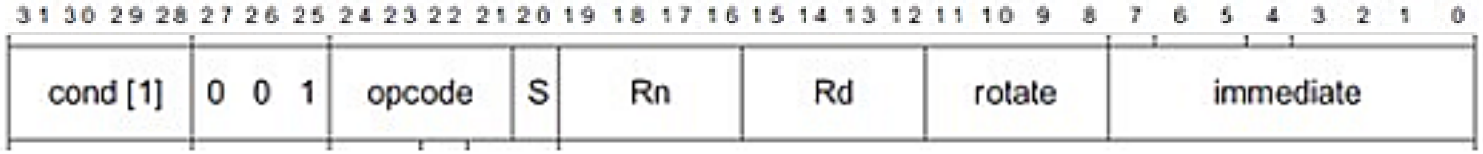
Istruzioni di data processing (ADD)

Istruzione: **ADD**

Il secondo operando può essere un registro:


```
add   r1, r2, r3 # r1 <= r2 + r3
```

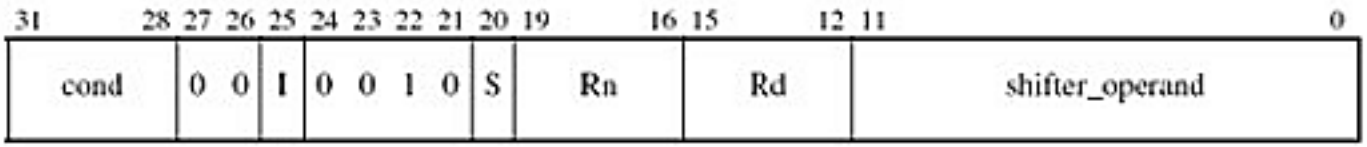
oppure una costante a 8 bit, il cui valore varia, quindi, fra 0 e 255:




```
add   r4, r5, #255 # r4 <= r5 + 255
```

Istruzioni di data processing (SUB)

Istruzione: **SUB**



SUB (Subtract) subtracts one value from a second value.

The second value comes from a register. The first value can be either an immediate value or a value from a register, and can be shifted before the subtraction.

SUB can optionally update the condition code flags, based on the result.

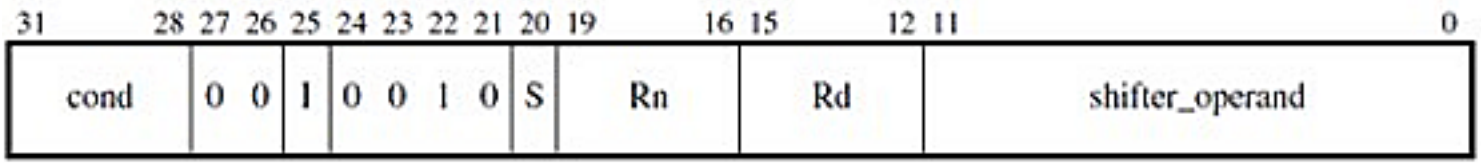
L'istruzione **SUB** sottrae il secondo operando dal primo e pone il risultato nel registro **Rd**.

```
SUB  R0, R1, R2 ; R0 = R1 - R2
SUB  R0, R1, #256 ; R0 = R1 - 256
SUBS  R0, R2, R3,LSL#1 ; R0 = R2 - (R3 << 1) and update flags
```

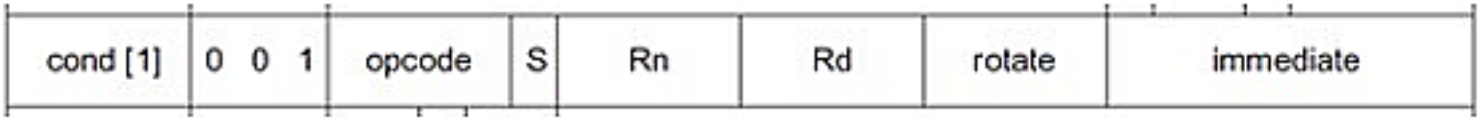
Istruzioni di data processing (SUB)

Istruzione: **SUB**

Il secondo operando può essere un registro:

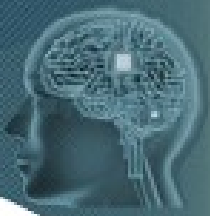


oppure una costante a 8 bit, il cui valore varia, quindi, fra 0 e 255:




```
sub   r4, r5, #255 # r4 <= r5 - 255
```

Istruzioni di data processing (Esempi)



Before:

r0 = 0x0000_0000
r1 = 0x0000_0002
r2 = 0x0000_0001

SUB r0, r1, r2

After:

r0 = 0x0000_0001

Before:

r0 = 0x0000_0000
r1 = 0x0000_0005

ADD r0, r1, r1, LSL#1

After:

r0 = 0x0000_000F

Istruzioni di data processing

Istruzioni di Moltiplicazione

La moltiplicazione di due numeri a 32 bit produce un valore a 64 bit.

L'architettura ARM fornisce istruzioni di moltiplicazione diverse, di cui alcune producono un valore a 32-bit, e altre a 64-bit.

- **MUL R1**, **R2**, **R3**– moltiplica due numeri a 32 bit e produce un risultato a 32 bit. Essa inserisce i bit meno significativi del prodotto in un registro e scarta i 32 più significativi.

- **UMULL R1**, **R2**, **R3**, **R4** (unsigned multiply long) – esegue una moltiplicazione senza segno di R3 e R4. I 32 bit meno significativi del prodotto sono inseriti in R1 e i 32 più significativi in R2.
- **SMULL R1**, **R2**, **R3**, **R4** (signed multiply long) – analoga a UMULL, ma con segno.

Ognuna di queste istruzioni ha anche una variante multiply-accumulate, **MLA**, **SMLAL**, e **UMLAL**, che aggiunge il prodotto ad un valore accumulato a 32 o 64 bit.

Il datapath con operazioni aritmetiche

Estendiamo il **datapath** per gestire le istruzioni di data processing **ADD**, **SUB**, **AND** e **ORR**, utilizzando la modalità di indirizzamento immediato.

In tal caso, le istruzioni hanno come operandi un registro ed una costante contenuta nei bit dell'istruzione stessa. L'**ALU** esegue l'operazione e il risultato viene scritto in un terzo registro.

Esse differiscono solo nella specifica operazione eseguita dall'**ALU**. Quindi, possono essere implementate tutte con lo stesso hardware utilizzando diversi segnali **ALUControl**.

I valori per **ALUControl** sono:

- **ADD** – **00**;
- **SUB** – **01**;
- **AND** – **10**;
- **ORR** – **11**.

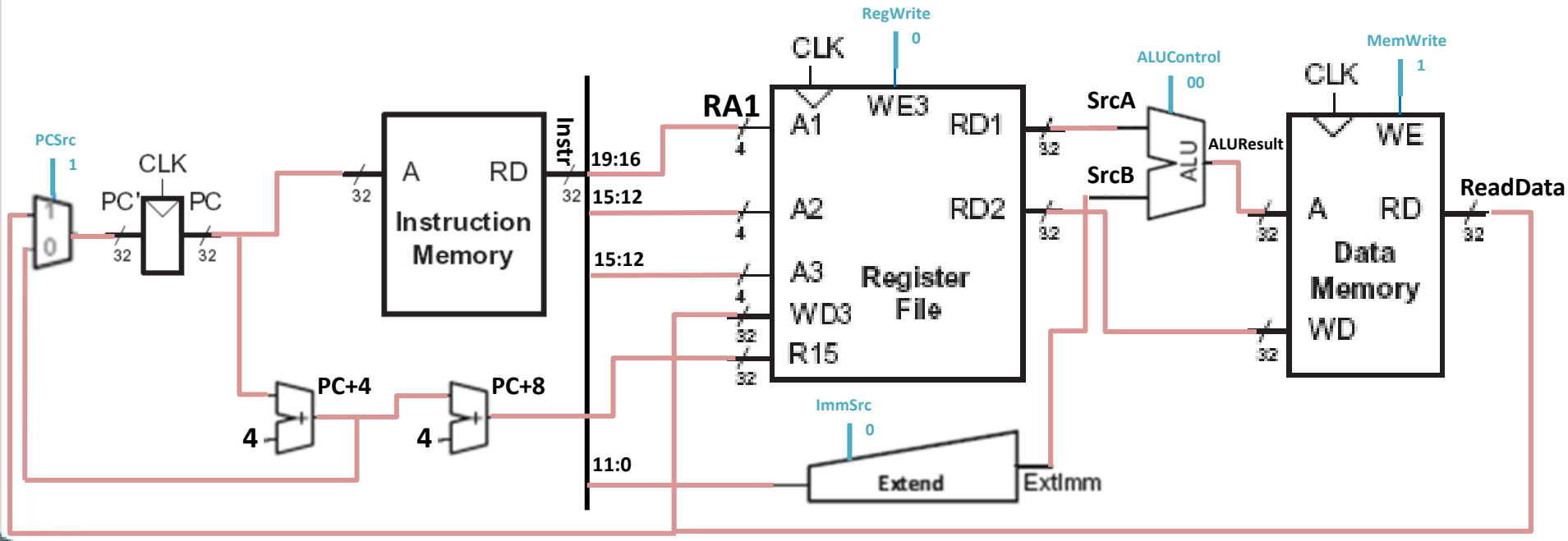
L'ALU imposta anche dei bit in **ALUFlags**$_{3:0}$:

- **Zero**;
- **Negativo**;
- **Carry**;
- **oVerflow**.

Il datapath con indirizzamento costante

Le istruzioni di data processing utilizzano costanti di **8** bit (non **12** bit), per cui il blocco **Extend** riceve in input un segnale di controllo **ImmSrc**:

- **ImmSrc** = **0** → **ExtImm** è esteso da $\mathbf{Instr}_{7:0}$;
- **ImmSrc** = **1** → **ExtImm** è esteso da $\mathbf{Instr}_{11:0}$ (per **LDR** o **STR**);



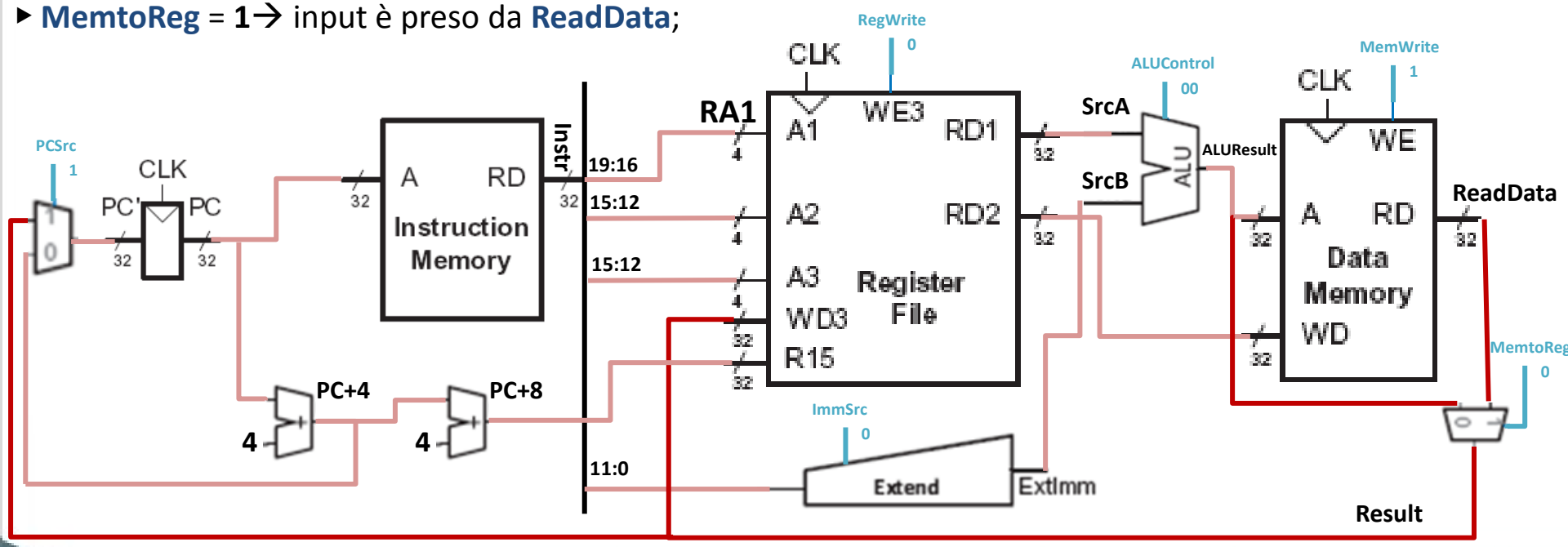
Il datapath con indirizzamento costante

Un altro aspetto da disambiguare riguarda la scrittura nel register file.
Esso può ricevere l'input sia dalla **memoria dati** (**LDR**), che dall'**ALU** (operazioni aritmetiche).

Aggiungiamo un altro **multiplexer** che permette di selezionare la sorgente di input tra **ReadData** e **ALUResult**. L'uscita del multiplexer è indicata con **Result**.

Il multiplexer richiede un segnale di controllo, ovvero **MemtoReg**.

- **MemtoReg** = **0** → input è preso da **ALUResult**;
- **MemtoReg** = **1**→ input è preso da **ReadData**;

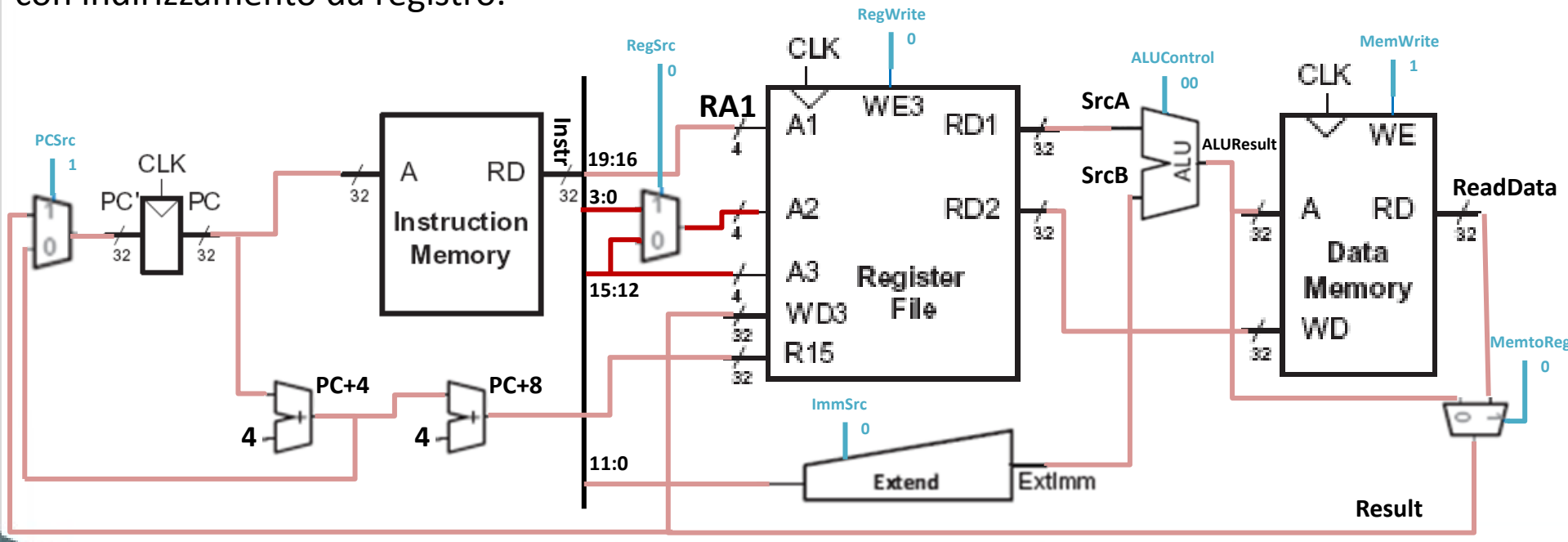


Il datapath con indirizzamento da registro

Le istruzioni di data processing con indirizzamento da registro ricevono la loro seconda fonte da **Rm**, specificato da **$Instr_{3:0}$**, piuttosto che da una costante.

Aggiungiamo un ulteriore **multiplexer** sugli ingressi del file registro. In base al valore del segnale di controllo **RegSrc** , **RA2** può essere selezionato fra:

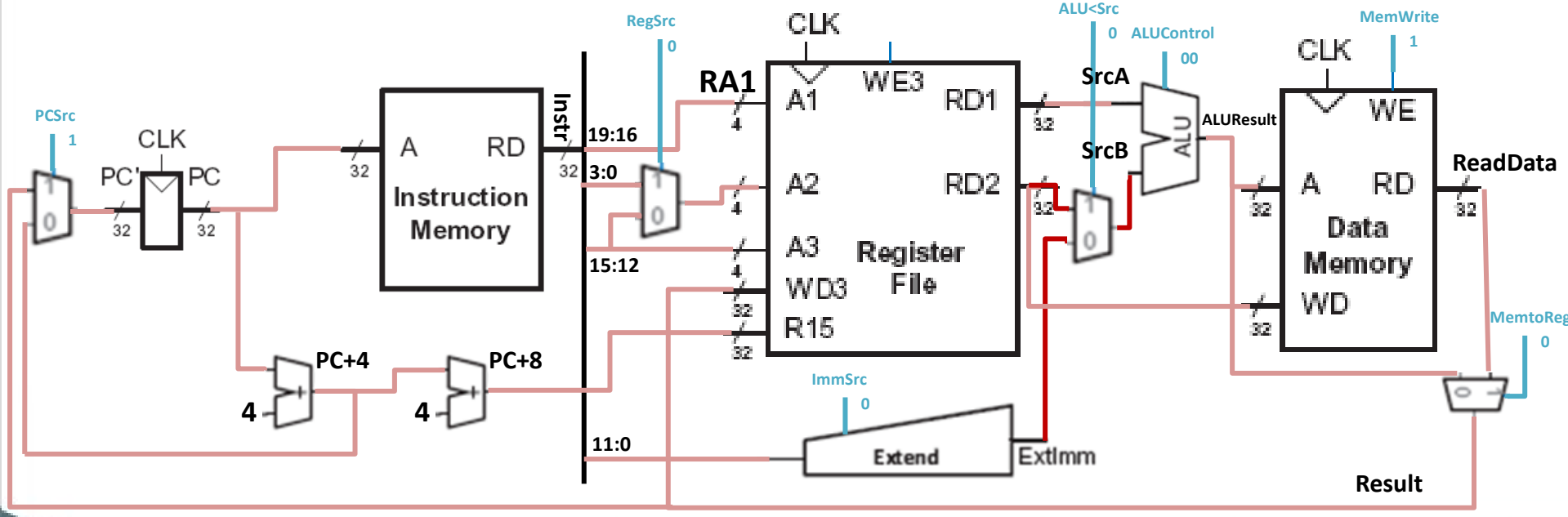
- **Rd** (**$Instr_{15:12}$**) per **STR**;
- **Rm** (**$Instr_{3:0}$**) per istruzioni di data processing con indirizzamento da registro.



Il datapath con indirizzamento da registro

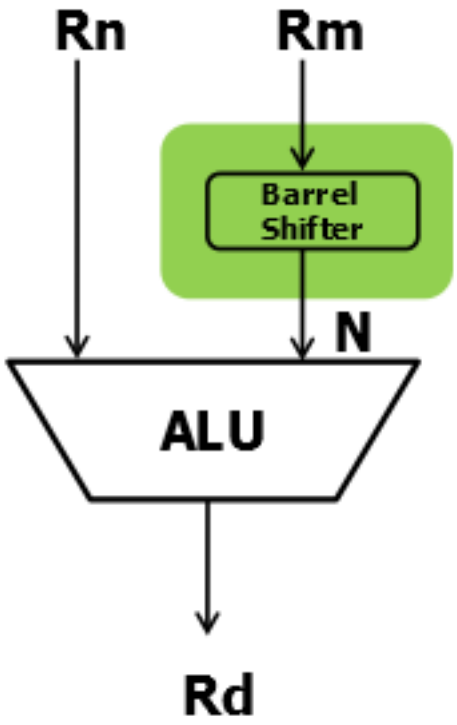
Aggiungiamo un ulteriore **multiplexer** sugli ingressi dell'**ALU** per selezionare questo secondo registro sorgente. In base al valore del segnale di controllo **ALUSrc**, la seconda sorgente della ALU viene selezionata tra:

- **ExtImm** per istruzioni, che utilizzano costanti;
- dal **register file** per istruzioni di data processing con indirizzamento da registro.



Istruzioni di data processing (confronto)

Nel linguaggio assembly del processore ARM, le istruzioni di confronto hanno la seguente sintassi.



Le istruzioni di confronto aggiornano i **flag** del registro **CPSR**, ma non modificano il contenuto di altri registri.

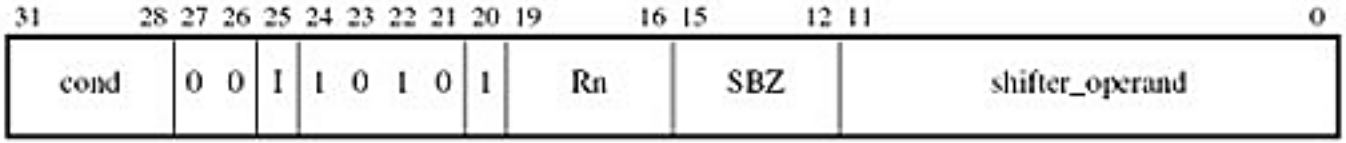
Dopo che i flag sono stati impostati, questa informazione può essere usata per modificare il flusso del programma attraverso le istruzioni condizionate.

Syntax: <instruction>{cond}{S} Rn, N

CMN	compare negated	*Flags set as a result of Rn + N*
CMP	Compare	*Flags set as a result of Rn – N*
TEQ	test for equality of two 32-bit values	*Flags set as a result of Rn ^ N*
TST	test bits of a 32-bit value	*Flags set as a result of Rn & N*

Istruzioni di data processing (CMP)

Istruzione: **CMP**



CMP (Compare) compares two values. The first value comes from a register. The second value can be either an immediate value or a value from a register, and can be shifted before the comparison.

CMP updates the condition flags, based on the result of subtracting the second value from the first.

L'istruzione **CMP** confronta i due operandi sottraendo il secondo dal primo. Si osservi che non vi è alcun registro destinazione, in quanto essa modifica unicamente i flag del registro CPSR.

```
CMP R0, R1;
```

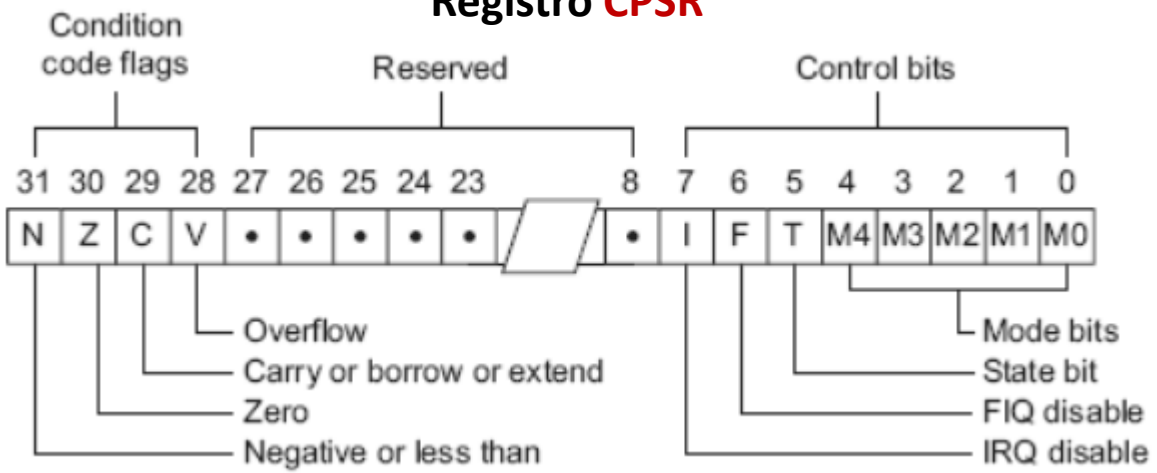
Istruzioni di data processing

Flag di Condizione

Le istruzioni ARM possono impostare dei flag di condizione sulla base del fatto che il risultato di una operazione sia negativo, zero, ecc. Le istruzioni successive eseguono a seconda dello stato di tali flag di condizione.

I flag di condizione in ARM, detti anche flag di stato, possono essere negativo (N), zero (Z), Carry (C), e overflow (V). Questi flag vengono impostati dalla ALU e si trovano nei 4 bit più significativi del Current Program Status Register (CPSR).

Registro CPSR



Il modo più comune per impostare i bit di stato è con l'istruzione di confronto (CMP), che sottrae il secondo operando dal primo e imposta i flag di condizione in base al risultato.

Esempio: **si supponga che un programma esegue CMP R4, R5, e poi ADDEQ R1, R2, R3.**

Se R4 e R5 sono uguali si imposta il flag Z, cosicché ADDEQ esegue solo se è impostato il flag Z. In linguaggio macchina, tale campo è indicato dal campo cond.

Istruzioni di data processing

Flag di Condizione

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\overline{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\overline{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\overline{V}$
1000	HI	Unsigned higher	$\overline{Z}C$
1001	LS	Unsigned lower or same	Z OR $\overline{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

	Unsigned	Signed
$A = 1001_2$	A = 9	A = –7
$B = 0010_2$	B = 2	B = 2

(a)

```
A – B:   1001      NZCV = 0011₂
       + 1110      HS: TRUE
       -------     GE: FALSE
         10111
```

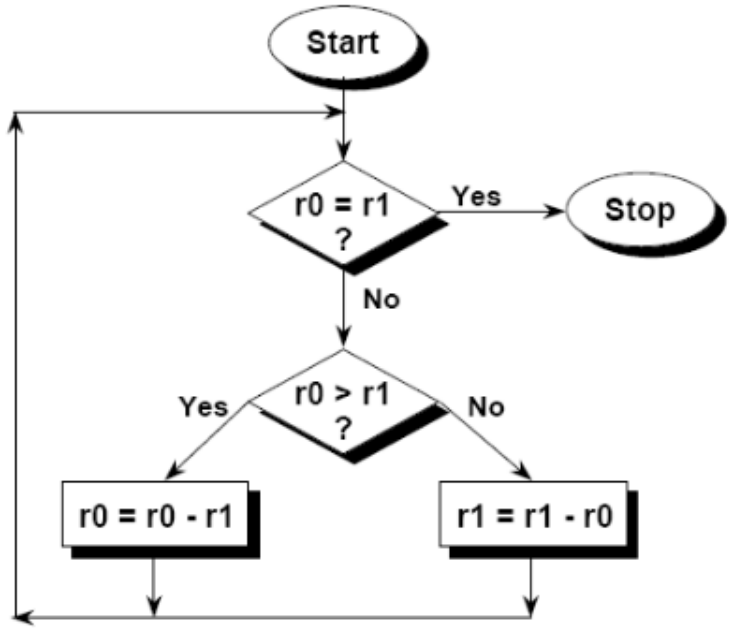
	Unsigned	Signed
$A = 0101_2$	A = 5	A = 5
$B = 1101_2$	B = 13	B = –3

(b)

```
A – B:   0101      NZCV = 1001₂
       + 0011      HS: FALSE
       -------     GE: TRUE
         1000
```

Istruzioni di data processing (condizioni)

Il processore ARM consente un codice molto più compatto.



1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N \oplus V})$

Processori convenzionali

```
MCD   cmp r0,r1          ;raggiunta la fine?
      beq FINE
      blt MIN            ; if r0 > r1 salta
      sub r0,r0,r1       ;r0 <- r0-r1
      b   MCD            ;altro giro
MIN   sub r1,r1,r0       ;r1 <- r1-r0
      b   MCD


FINE
```

Processore ARM

```
MCD   cmp r0,r1          ;if r0 > r1
      subgt r0,r0,r1     ;then r0 <- r0-r1
      sublt r1,r1,r0     ;else r1 <- r1-r0
      bne MCD            ;raggiunta la fine?
```

Istruzioni di data processing

Istruzioni di Shift

Queste istruzioni shiftano il valore in un registro da sinistra a destra, scartando i bit oltre quello meno significativo. L'operazione di **rotate** ruota il valore in un registro a destra fino a 31 bit. Tanto le operazioni di scorrimento, quanto quella di rotazione sono genericamente indicate come operazioni di scorrimento.

In ARM le operazioni di scorrimento sono:

- **LSL** (shift logico a sinistra);
- **LSR** (shift logico a destra);
- **ASR** (shift aritmetico a destra);
- **ROR** (rotazione a destra).

	Source register			
R5	1111 1111	0001 1100	0001 0000	1110 0111

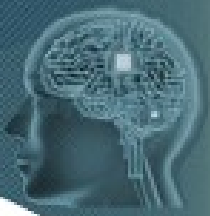
Assembly Code		Result			
LSL R0, R5, #7	R0	1000 1110	0000 1000	0111 0011	1000 0000
LSR R1, R5, #17	R1	0000 0000	0000 0000	0111 1111	1000 1110
ASR R2, R5, #3	R2	1111 1111	1110 0011	1000 0010	0001 1100
ROR R3, R5, #21	R3	1110 0000	1000 0111	0011 1111	1111 1000

Non vi è alcuna istruzione ROL perché la rotazione a sinistra può essere eseguita con una rotazione a destra di una quantità complementare.

	Source registers			
R8	0000 1000	0001 1100	0001 0110	1110 0111
R6	0000 0000	0000 0000	0000 0000	0001 0100

Assembly code		Result			
LSL R4, R8, R6	R4	0110 1110	0111 0000	0000 0000	0000 0000
ROR R5, R8, R6	R5	1100 0001	0110 1110	0111 0000	1000 0001

Istruzioni di data processing



Istruzioni di Shift

Uno **shift a sinistra** riempie sempre i bit meno significativi con uno 0.

Uno **shift a destra** può essere sia **logico** (bit più significativi impostati a 0) o **aritmetico** (bit più significativi pari al bit di segno).

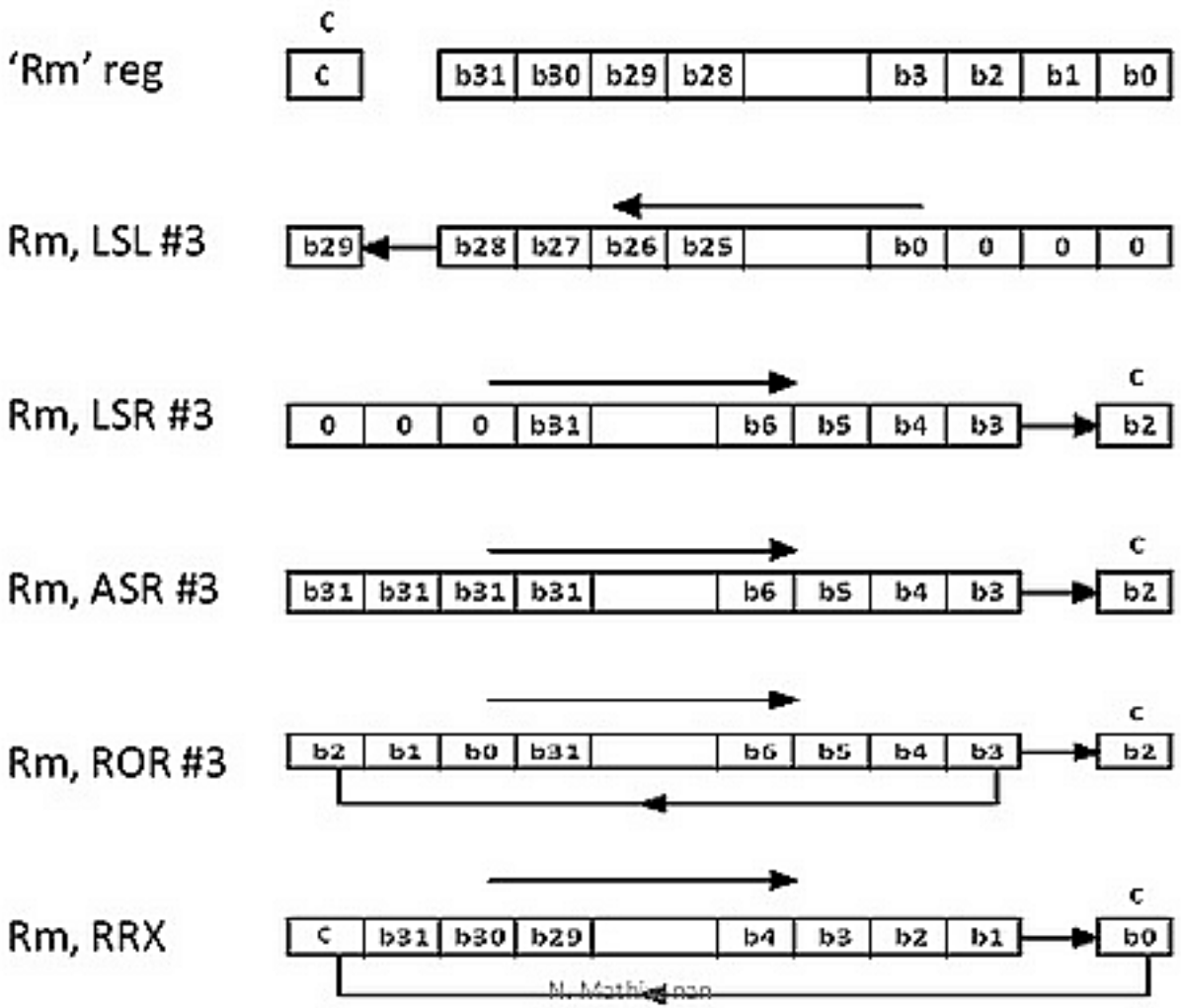
Il valore dello spostamento può essere una costante o un registro.

Lo shift di un valore a sinistra di **N** posizioni è equivalente a moltiplicare tale valore per 2^N. Analogamente, uno shift a destra di un valore **N** è equivalente a dividere per 2^N.

Gli shift logici sono spesso utilizzati per estrarre o assemblare insiemi di bit.

Istruzioni di data processing

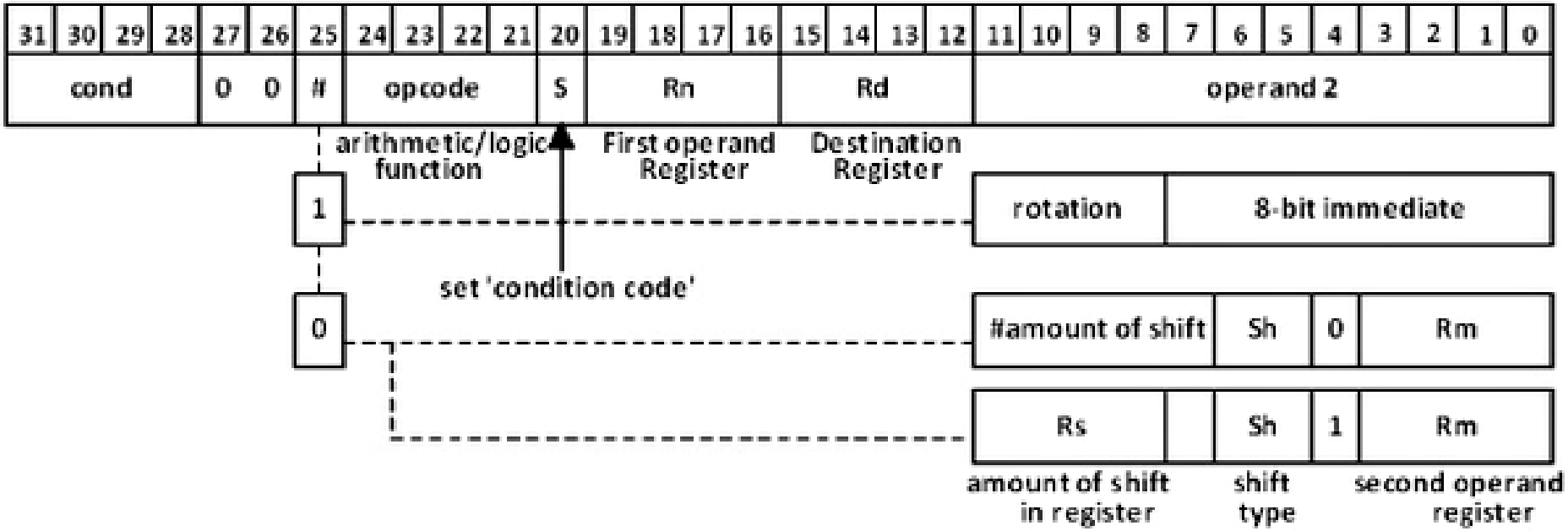
Esempi di operazionid di shift:



Istruzioni di data processing

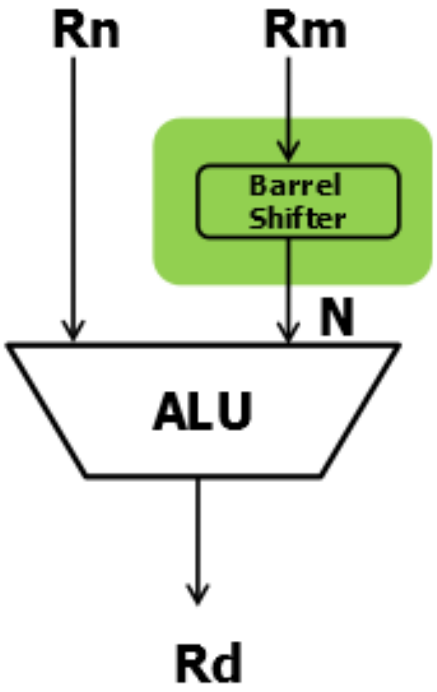
Sintesi schematica

L'insieme delle istruzioni di data processing può essere riassunto nel seguente schema:



Istruzioni di data processing (spostamento)

Nel linguaggio assembly del processore ARM, le istruzioni di spostamento hanno la seguente sintassi.

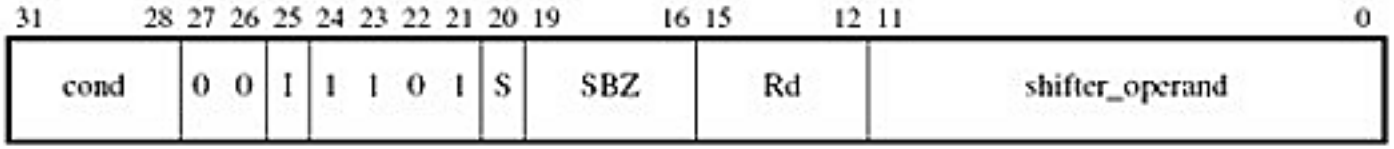


Syntax: <instruction>{cond}{S} Rd, N

MOV	Move a 32-bit value into a register	*Rd = N*
MVN	Move the NOT of the 32-bit value into a register	*Rd = ~ N*

Istruzioni di data processing (MOV)

Istruzione: **MOV**



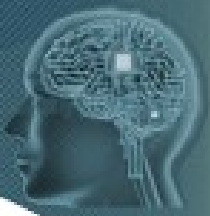
MOV (Move) writes a value to the destination register. The value can be either an immediate value or a value from a register, and can be shifted before the write.

MOV can optionally update the condition code flags, based on the result.

L'istruzione **MOV** copia nel registro destinazione il valore contenuto in un altro registro o il valore di una costante. Essa è molto utile per copiare valori fra registri o per impostare il valore iniziale di un registro.

```
MOV R0, R0; move R0 to R0, Thus, no effect
MOV R0, R0, LSL#3 ;  R0 = R0 * 8
MOV PC, R14; (R14: link register)   Used to return to caller
MOVS PC, R14; PC <- R14 (lr), CPSR <- SPSR
                ;  Used to return from interrupt or exception
```

Istruzioni di data processing (Esempi)



Before: cpsr = nzcv

r0 = 0x0000_0000

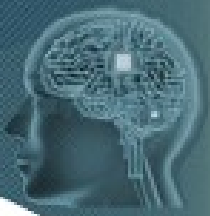
r1 = 0x8000_0004

MOVS r0, r1, LSL #1

After:

r0 = 0x0000_0008

Istruzioni di data processing



MOV	Move a 32-bit value	MOV Rd,n	Rd = n
MVN	Move negated (logical NOT) 32-bit value	MVN Rd,n	Rd = ~n
ADD	Add two 32-bit values	ADD Rd,Rn,n	Rd = Rn+n
ADC	Add two 32-bit values and carry	ADC Rd,Rn,n	Rd = Rn+n+C
SUB	Subtract two 32-bit values	SUB Rd,Rn,n	Rd = Rn-n
SBC	Subtract with carry of two 32-bit values	SBC Rd,Rn,n	Rd = Rn-n+C-1
RSB	Reverse subtract of two 32-bit values	RSB Rd,Rn,n	Rd = n-Rn
RSC	Reverse subtract with carry of two 32-bit values	RSC Rd,Rn,n	Rd = n-Rn+C-1
AND	Bitwise AND of two 32-bit values	AND Rd,Rn,n	Rd = Rn AND n
ORR	Bitwise OR of two 32-bit values	ORR Rd,Rn,n	Rd = Rn OR n
EOR	Exclusive OR of two 32-bit values	EOR Rd,Rn,n	Rd = Rn XOR n
BIC	Bit clear. Every '1' in second operand clears corresponding bit of first operand	BIC Rd,Rn,n	Rd = Rn AND (NOT n)
CMP	Compare	CMP Rd,n	Rd-n & change flags only
CMN	Compare Negative	CMN Rd,n	Rd+n & change flags only
TST	Test for a bit in a 32-bit value	TST Rd,n	Rd AND n, change flags
TEQ	Test for equality	TEQ Rd,n	Rd XOR n, change flags

MUL	Multiply two 32-bit values	MUL Rd,Rm,Rs	Rd = Rm*Rs
MLA	Multiple and accumulate	MLA Rd,Rm,Rs,Rn	Rd = (Rm*Rs)+Rn

Istruzioni di branch (o salto)

Branching instruction

Un programma di solito esegue in sequenza, incrementando il Program Counter (PC) di 4 (32 bit) dopo ciascuna istruzione, in modo da puntare alla successiva istruzione.

Le istruzioni Branch permettono di cambiare il valore del PC. ARM include due tipi di branch: simple branch (B) e branch and link (BL).

Come altre istruzioni ARM, i branch possono essere condizionati o incondizionati.

Il codice assembly utilizza le etichette per indicare i blocchi di istruzioni nel programma.

Quando il codice assembly è tradotto in codice macchina, queste etichette vengono tradotte in indirizzi di istruzione.

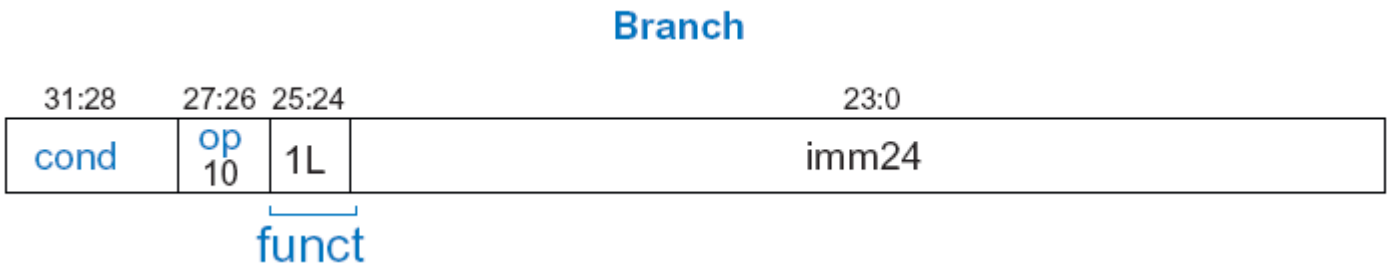
Istruzioni di Branch

Le istruzioni di **Branch** utilizzano un unico operando costante di 24 bit, **imm24**.

Esse hanno un campo **cond** di 4 bit e un campo **op** di 2 bit, il cui valore è 10_2.

Il campo **funct** ha solo 2 bit. Il bit più significativo è sempre 1 per i branch. Il bit meno significativo, **L**, indica il tipo di operazione di branch: 1 per **BL** e 0 per **B**.

I restanti 24 bit, **imm24**, rappresentano un valore in complemento a due, che specifica la posizione dell'istruzione relativamente all'indirizzo **PC** + 8.



Il datapath per i branch

L'istruzione di salto somma una costante a **24**-bit a **PC+8** e scrive il risultato di nuovo nel **PC**.

La costante viene moltiplicata per 4 ed estesa con segno. Pertanto, la logica **Extend** necessita di una ulteriore modalità. **ImmSrc** è, quindi, esteso a 2 bit.

ImmSrc	ExtImm	Description
00	{24 0s} $Instr_{7:0}$	8-bit unsigned immediate for data-processing
01	{20 0s} $Instr_{11:0}$	12-bit unsigned immediate for LDR/STR
10	{6 $Instr_{23}$} $Instr_{23:0}$ 00	24-bit signed immediate multiplied by 4 for B

Istruzioni di Branch

L'istruzione **BL** (Branch and Link) è usata per la chiamata di una subroutine

- Salva l'indirizzo di ritorno (**R15**) in **R14**
- Il ritorno dalla routine si effettua copiando **R14** in **R15**:

MOV R15, R14

Il registro **R14** ha la funzione (architetturale) di subroutine Link Register (**LR**).

In esso viene salvato l'indirizzo di ritorno (ovvero il contenuto del registro **R15**) quando viene eseguita l'istruzione **BL** (Branch and Link).

```
    ....
    BL    function          ; call 'function'
    ....                    ; procedure returns to here
    ....
function                    ; function body
    ....
    ....
    MOV   PC, LR            ; Put R14 into PC to return
```

Il datapath completo

Dato che **PC+8** è letto dalla prima porta del register file, è necessario un **multiplexer** per selezionare **R15** come ingresso di **RA1**. Il multiplexer è controllato dal segnale **RegSrc**, il cui valore è preso dai bit $\mathbf{Instr}_{19:16}$, per la maggior parte delle istruzioni ed è impostato a **15** per le istruzioni di branch (**B**).

MemtoReg è impostato a **0** e **PCSrc** è impostato a **1** per selezionare il nuovo **PC** da **ALUResult**.