Architettura degli Elaboratori I - B

Unità di Controllo e Analisi delle Prestazioni

Architettura a ciclo singolo

Daniel Riccio/Alberto Aloisio

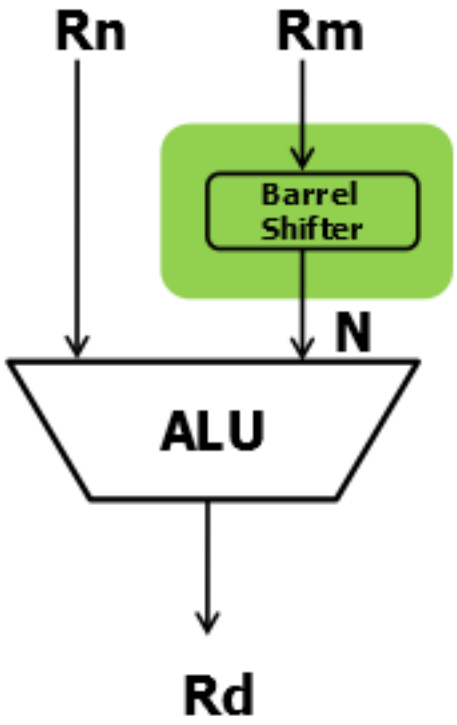
Università di Napoli, Federico II

4 aprile 2018

Rif. Capitolo 1
ARM System-on-Chip Architecture, 2Ed, Addison Wesley

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.

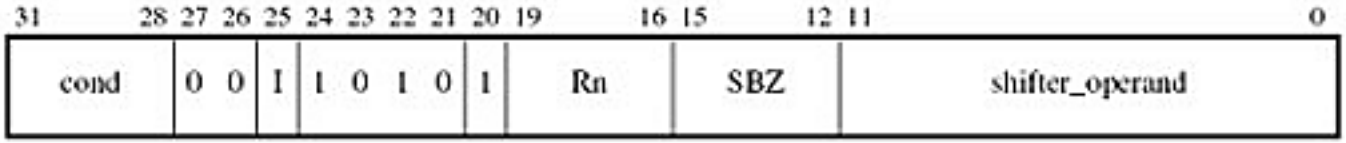
Dopo che i flag sono stati impostati, questa informazione può essere usata per modificare il flusso del programma attraverso le istruzioni condizionate.

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;
```

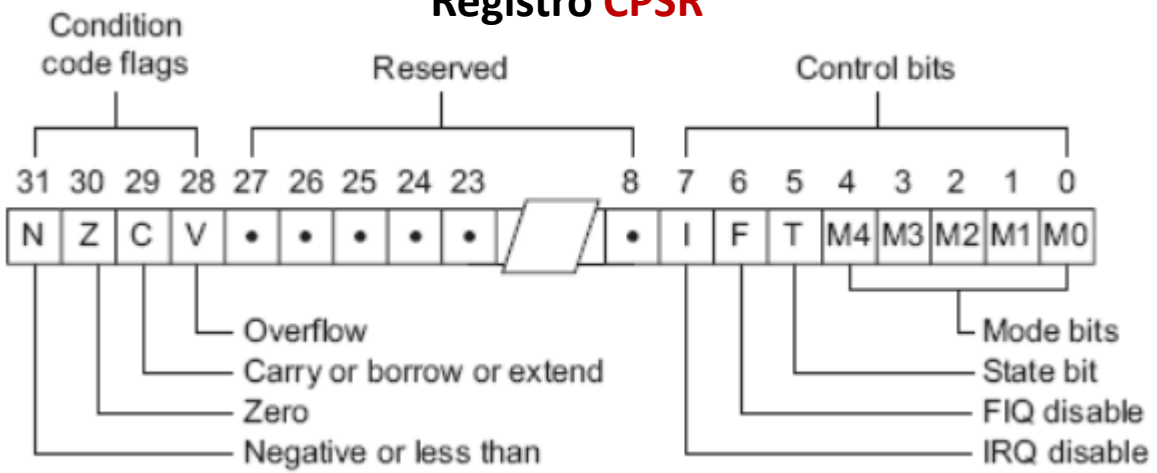
Istruzioni di data processing

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

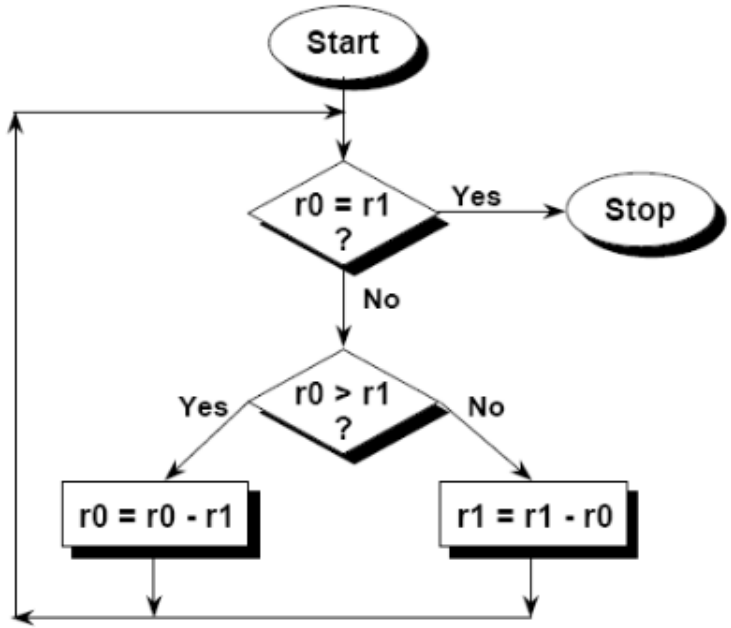
```
A – B:   1001      NZCV = 0011₂
       + 1110      HS: TRUE
(a)      10111     GE: FALSE
```

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

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

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

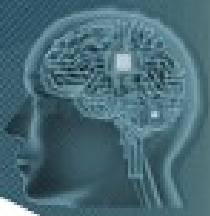
Assembly Code		Result			
LSL R0, R5, #7	R0	1000 1110	0000 1000	0111 0011	1000 0000
LSR R1, R5, #17	R1	0000 0000	0000 0000	0111 1111	1000 1110
ASR R2, R5, #3	R2	1111 1111	1110 0011	1000 0010	0001 1100
ROR R3, R5, #21	R3	1110 0000	1000 0111	0011 1111	1111 1000

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).

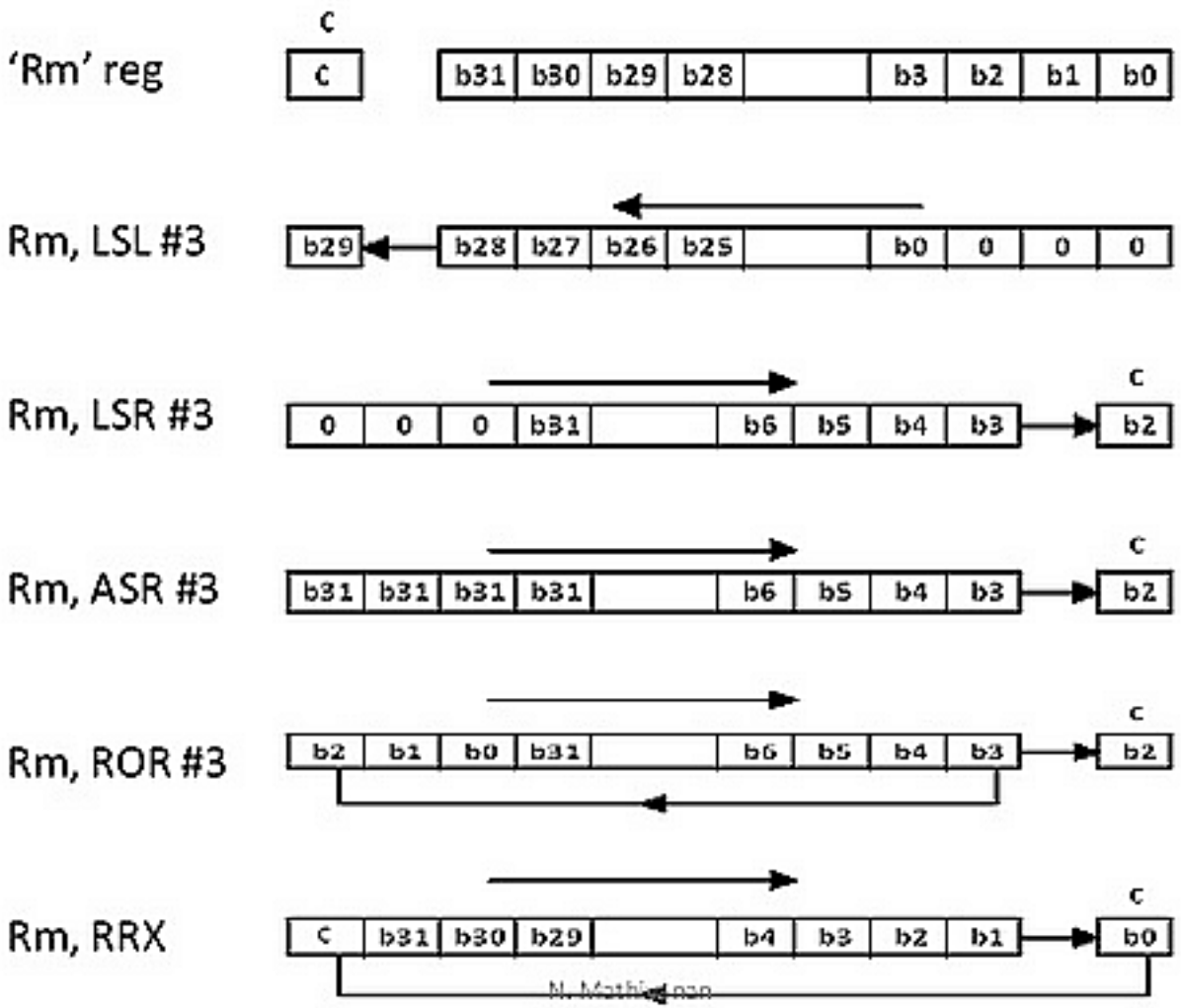
Il valore dello spostamento può essere una costante o un registro.

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

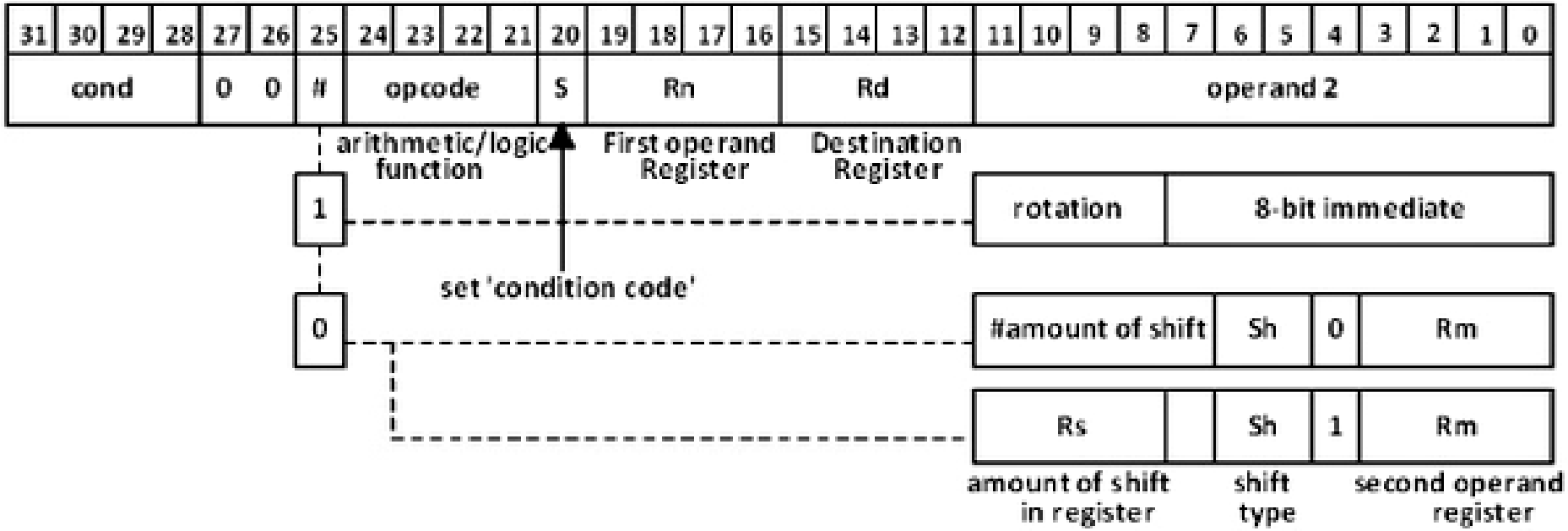
Esempi di operazionid di shift:



Istruzioni di data processing

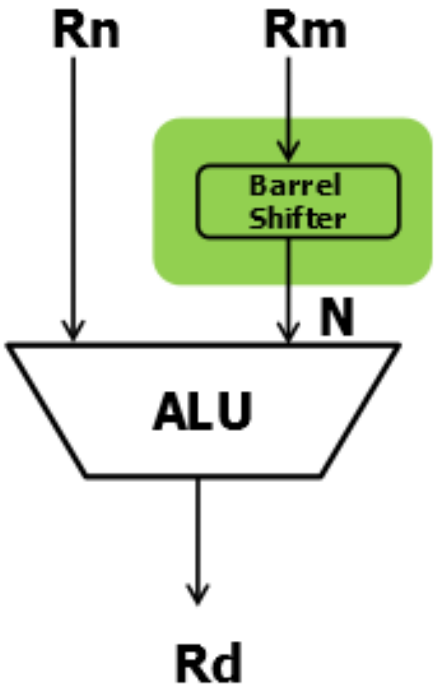
Sintesi schematica

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.

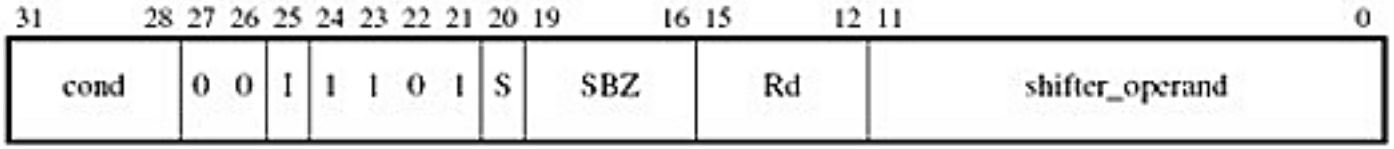


Syntax: <instruction>{cond}{S} Rd, N

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

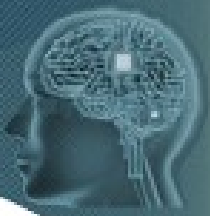
r1 = 0x8000_0004

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

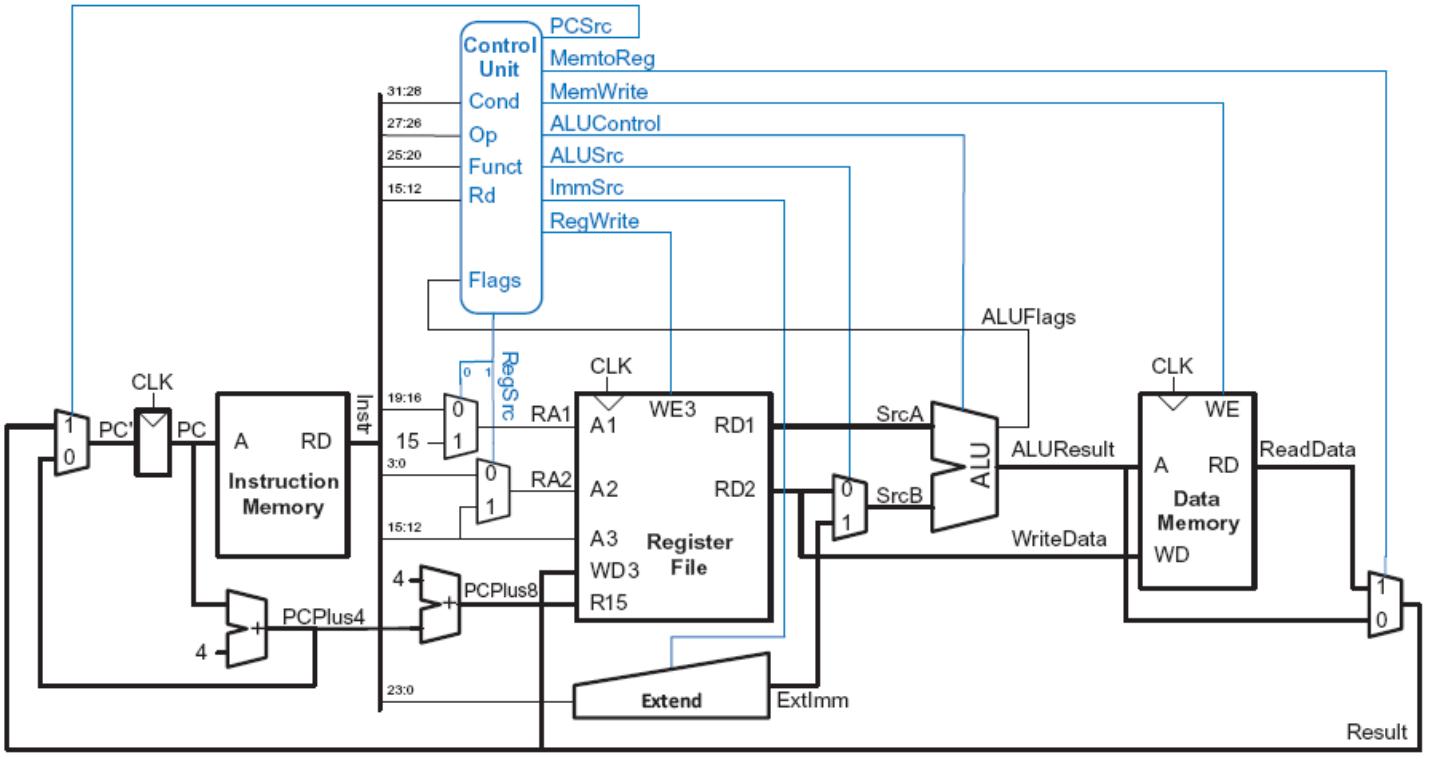
MUL	Multiply two 32-bit values	MUL Rd,Rm,Rs	Rd = Rm*Rs
MLA	Multiple and accumulate	MLA Rd,Rm,Rs,Rn	Rd = (Rm*Rs)+Rn

L'unità di controllo

L'unità di controllo calcola i segnali di controllo in base a:

- i campi **cond**, **op**, e **funct** dell'istruzione (**$Instr_{31:28}$**, **$Instr_{27:26}$**, e **$Instr_{25:20}$**);
- i **flag**;
- se il registro di destinazione è il **PC**.

Il controller memorizza anche i **flag di stato** attuali e li aggiorna in modo appropriato.



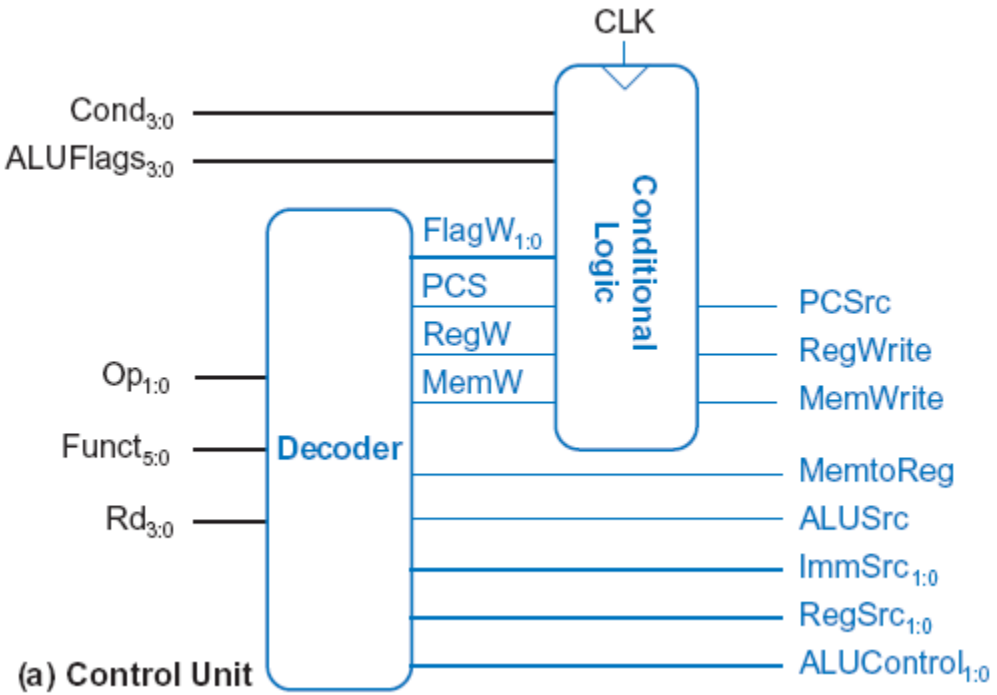
L’unità di controllo

Dividiamo l’unità di controllo in due parti principali:

- il **decoder** – genera i segnali di controllo sulla base dei campi di Instr;
- la **logica condizionale** – gestisce i flag di stato e li aggiorna quando l’istruzione deve essere eseguita su condizione.

Il **Decoder** è composto da:

- un **decodificatore principale**, che produce la maggior parte dei segnali di controllo;
- un **decoder ALU**, che utilizza il campo Funct per determinare il tipo di istruzione data-processing;
- la **logica di controllo del PC**, che determina se il PC deve essere aggiornato a causa di una istruzione di branch o di una scrittura in R15.



L’unità di controllo

Il **Decoder principale** ha come compiti:

- determinare il tipo di istruzione: data processing con registro o costante, STR, LDR, o B.

- produrre i segnali di controllo adeguati per il datapath. Alcuni segnali sono inviati direttamente al datapath: **MemtoReg**, **ALUSrc**, **ImmSrc1:0**, e **RegSrc1:0**.

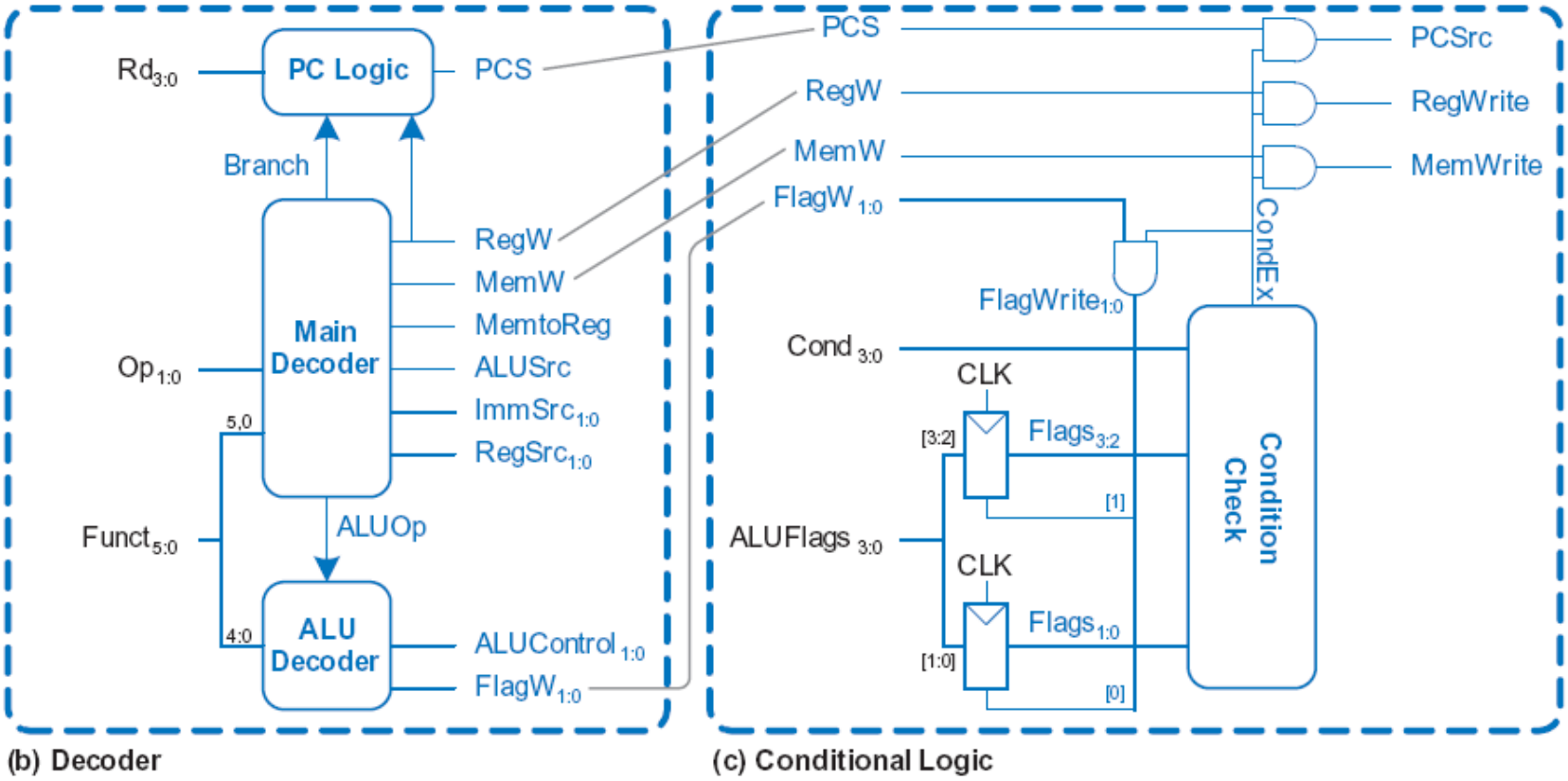
- generare i segnali che abilitano la scrittura (**MemW** e **RegW**), i quali devono passare attraverso la logica condizionale prima di diventare segnali datapath (**MemWrite** e **RegWrite**). Tali segnali possono essere azzerati dalla logica condizionale, se la condizione non è soddisfatta.

- generare i segnali **Branch** e **ALUOp**, utilizzati rispettivamente per indicare l'istruzione B o il tipo di istruzione data processing.

La logica per il decoder principale può essere sviluppata dalla tabella di verità utilizzando le tecniche standard per la progettazione della logica combinatoria.

L’unità di controllo

I segnali che abilitano la scrittura (**MemW** and **RegW**), i quali devono passare attraverso la logica condizionale prima di diventare segnali datapath (**MemWrite** e **RegWrite**). Tali segnali possono essere azzerati dalla logica condizionale, se la condizione non è soddisfatta.



L'unità di controllo

La tabella di verità per le funzioni logiche del decodificatore principale è:

Table 7.2 Main Decoder truth table

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

L'unità di controllo

Il **decoder ALU** ha come compiti:

- Impostare il segnale **ALUControl** in base al tipo di istruzione (**ADD**, **SUB**, **AND**, **ORR**);
- Impostare **FlagW** per aggiornare i flag di stato quando è impostato il bit **S**-bit.

Sapendo che **ADD** e **SUB** aggiornano tutti i flag, mentre **AND** e **ORR** aggiornano solo i flag **N** e **Z**, osserviamo che sono necessari due bit di **FlagW**:

- **FlagW1** per l'aggiornamento di **N** e **Z** (**$Flags_{3:2}$**)
- **FlagW0** per l'aggiornamento di **C** e **V** (**$Flags_{1:0}$**).

$FlagW_{1:0}$ viene azzerato dalla logica condizionale quando la condizione non è soddisfatta (**Condex** = 0).

Table 7.3 ALU Decoder truth table

ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00 (Add)	00
1	0100	0	ADD	00 (Add)	00
		1			11
	0010	0	SUB	01 (Sub)	00
		1			11
	0000	0	AND	10 (And)	00
		1			10
	1100	0	ORR	11 (Or)	00
		1			10

L’unità di controllo

La **logica del PC** controlla se l'istruzione è una scrittura in **R15** o un branch secondo la condizione:

$$\mathbf{PCS} = ((\mathbf{Rd} == \mathbf{15}) \,\&\, \mathbf{RegW}) \mid \mathbf{Branch})$$

I bit **PCS** possono essere azzerati dalla logica condizionale prima di essere inviati al datapath come **PCSrc**.

La logica condizionale determina se l'istruzione deve essere eseguita (**CondEx**) in base al campo **cond** ed ai valori correnti dei flag **N**, **Z**, **C**, e **V** ($\mathbf{Flags}_{3:0}$).

Se l’istruzione non deve essere eseguita, i **segnali di scrittura** e **PCSrc** sono impostati a **0**, in modo che l'istruzione non cambi lo stato architettonico.

La logica condizionale aggiorna anche alcuni o tutti gli **ALUFlags** quando **FlagW** viene impostato dal **decoder ALU** e la condizione della istruzione è soddisfatta (**Condex** = 1).

L'unità di controllo

La tabella dei valori per il segnale **CondEx** è:

Table 6.3 Condition mnemonics

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

Analisi delle prestazioni

Ogni istruzione nel processore a ciclo singolo impiega un ciclo di clock, quindi il CPI è 1.

I critical path per l'istruzione LDR sono:

- (t_{pcq_PC}) – caricamento di un nuovo indirizzo (PC) sul fronte di salita del clock;
- (t_{mem}) – lettura dell'istruzione in memoria;
- (t_{dec}) – il Decoder principale calcola RegSrc0, che induce il multiplexer a scegliere $Instr_{19:16}$ come RA1, e il register file legge questo registro come srcA;
- ($\max[t_{mux}+t_{RFread}, t_{ext}+t_{mux}]$) – mentre il register file viene letto, il campo costante viene esteso e viene selezionata dal multiplexer ALUSrc per determinare srcB.
- (t_{ALU}) – l'ALU somma srcA e srcB per trovare l'indirizzo effettivo.
- (t_{mem}) – La memoria di dati legge da questo indirizzo.
- (t_{mux}) – il multiplexer MemtoReg seleziona ReadData.
- ($t_{RFsetup}$) – viene impostato il segnale Result ed il risultato viene scritto nel register file.

Analisi delle prestazioni

Il tempo totale è dato dalla somma dei parziali:

$$T_{c1} = t_{pcq\_PC}+t_{mem}+t_{dec}+\max[t_{mux}+t_{RFread}, t_{ext}+t_{mux}]+t_{ALU}+t_{mem}+t_{mux}+t_{RFsetup};$$

Nella maggior parte delle implementazioni, l'ALU, la memoria ed il register file sono sostanzialmente più lenti di altri blocchi combinatori. Pertanto, il tempo di ciclo può essere semplificato come:

$$T_{c1} = t_{pcq\_PC}+2t_{mem}+t_{dec}+t_{RFread}+t_{ALU}+2t_{mux}+t_{RFsetup};$$

Table 7.5 Delay of circuit elements

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Analisi delle prestazioni

Domanda: **qual è il tempo di esecuzione per un programma con 100 miliardi di istruzioni?**

Risposta:
secondo l'equazione

$$T_{c1} = t_{pcq\_PC} + 2t_{mem} + t_{dec} + t_{RFread} + t_{ALU} + 2t_{mux} + t_{RFsetup}$$

il tempo di ciclo del processore singolo ciclo è

Tc1 = 40 + 2 (200) + 70 + 100 + 120 + 2 (25) + 60
= 840 ps.

Table 7.5 Delay of circuit elements

Element	Parameter	Delay (ps)
Register clk-to-Q	t_{pcq}	40
Register setup	t_{setup}	50
Multiplexer	t_{mux}	25
ALU	t_{ALU}	120
Decoder	t_{dec}	70
Memory read	t_{mem}	200
Register file read	t_{RFread}	100
Register file setup	$t_{RFsetup}$	60

Secondo l'equazione

$$Tempo\ di\ esecuzione = (\#istruzioni)\left(\frac{cicli}{istruzione}\right)\left(\frac{secondi}{ciclo}\right)$$

il tempo di esecuzione totale è

T1 = (100 × 10^9 istruzioni) (1 ciclo / di istruzione)
(840 × 10^{-12} s / ciclo) = 84 secondi.