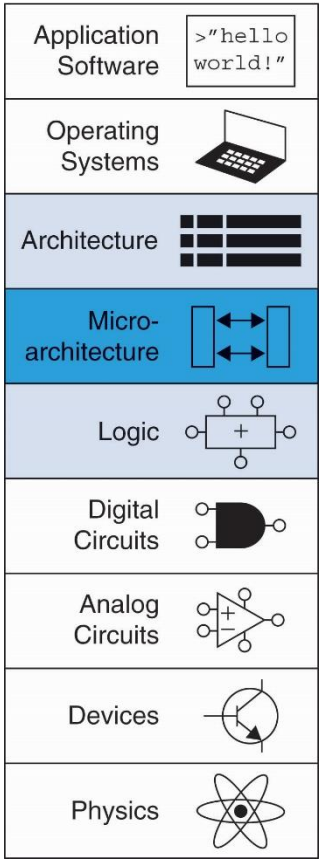
Microarchittetura ARM

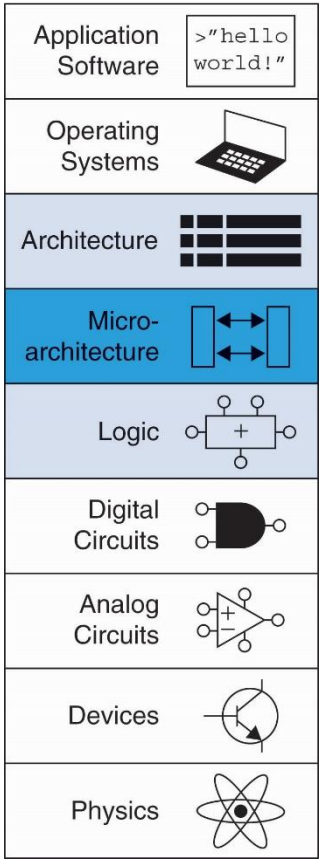
Architettura: rappresenta la struttura di un calcolatore dal punto di vista di un programmatore

- È definita dal set di istruzioni e operandi che costituisce il *linguaggio macchina*
- Al linguaggio macchina corrisponde un linguaggio *assembly*
  - **Assembly language:** formato human-readable delle istruzioni
  - **Machine language:** formato computer-readable (1 e 0)
- Esistono molte architetture differenti
  - **ARM**
  - **X86**
  - **MIPS**
  - **SPARC**
  - **PowerPC**



Microarchitettura: definisce la disposizione specifica di registri, ALU, macchine a stati finiti, le memorie e altri blocchi logici necessari per implementare una architettura

- Come due algoritmi di ordinamento (bubble e quick sort) realizzano la stessa funzione con procedure differenti, così due microarchitetture possono realizzare la medesima architettura ma soluzioni tecnologiche e prestazioni molto differenti
- Ad esempio Intel e AMD realizzano diversi modelli (microarchitetture) della medesima architettura x86.



Architettura ARM

- Sviluppata negli anni 80 dalla Advanced RISC Machines – ora chiamata ARM Holdings
- Circa 10 miliardi di processor ARM venduti ogni anno
- Impiegata soppratutto nel settore degli smartphone e tablet
- Circa il 75% della popolazione umana usa prodotti equipaggiati da un processore ARM

Design principles

L'architettura ARM segue quattro principi di progettazione fondamentali:

1. **Regularity supports design simplicity**
2. **Make the common case fast**
3. **Smaller is faster**
4. **Good design demands good compromises**

Regularity support simplicity

C Code

```
a = b + c;
```

ARM Assembly Code

```
ADD a, b, c
```

C Code

```
a = b - c;
```

ARM Assembly Code

```
SUB a, b, c
```

ADD: mnemonico – indica l'operazione da eseguire

b, c: operandi sorgente (source operands)

a: operando destinazione (destination operand)

Regularity supports design simplicity

- Ogni istruzione è rappresenta da una parola di 32 bit (anche se alcune istruzioni richiederebbero meno di 32 bit di codifica, gestire istruzioni di lunghezza variabile aumenterebbe la complessità)
- I formati sono consistenti, nell'esempio precedente stesso numero di operandi (due sorgenti e una destinazione)
- *Questo facilita l'encoding in hardware*

Make the common case fast

- L'architettura ARM comprende solo istruzioni semplice e di uso frequente
- Hardware necessario per decodificare ed eseguire le istruzioni è così *semplice* e veloce
- Istruzioni complesse (che sono meno comuni) corrispondono a più istruzioni ARM

C Code

```
a = b + c - d;
```

ARM assembly code

```
ADD t, b, c  ; t = b + c
SUB a, t, d  ; a = t - d
```

- ARM è un **Reduced Instruction Set Computer (RISC)**: poche istruzioni e semplici
- X86 è un **Complex Instruction Set Computers (CISC)**: molte istruzioni complesse

Smaller is faster

I dati sono memorizzati:

- Direttamente all'interno di una istruzione.
  - Costanti (anche dette *immediates*)
- In registri
  - ARM ha solo 16 registri
  - I registri sono più veloci della memoria
  - Ogni registro è constituito da 32 bits
  - ARM chiamata una architettura a "32-bit" perché opera su dati di 32-bit
- In memoria
  - Più lenta dei registi ma più capiente
  - A sua volta suddivisa in diversi livelli: cache, centrale, di massa
  - le istruzioni ARM operano esclusivamente sui registri, quindi i dati residenti nella memoria devono essere spostati in un registro prima di poter essere elaborati.

Good design demands good compromises

Al fine di massimizzare la semplicità sarebbe preferibile anche avere un unico formato per le istruzioni. Tuttavia, questa scelta risulterebbe troppo restrittiva.

Il compromesso scelto da ARM (esempio del principio 4) è di avere **tre** possibili **formati di istruzione**:

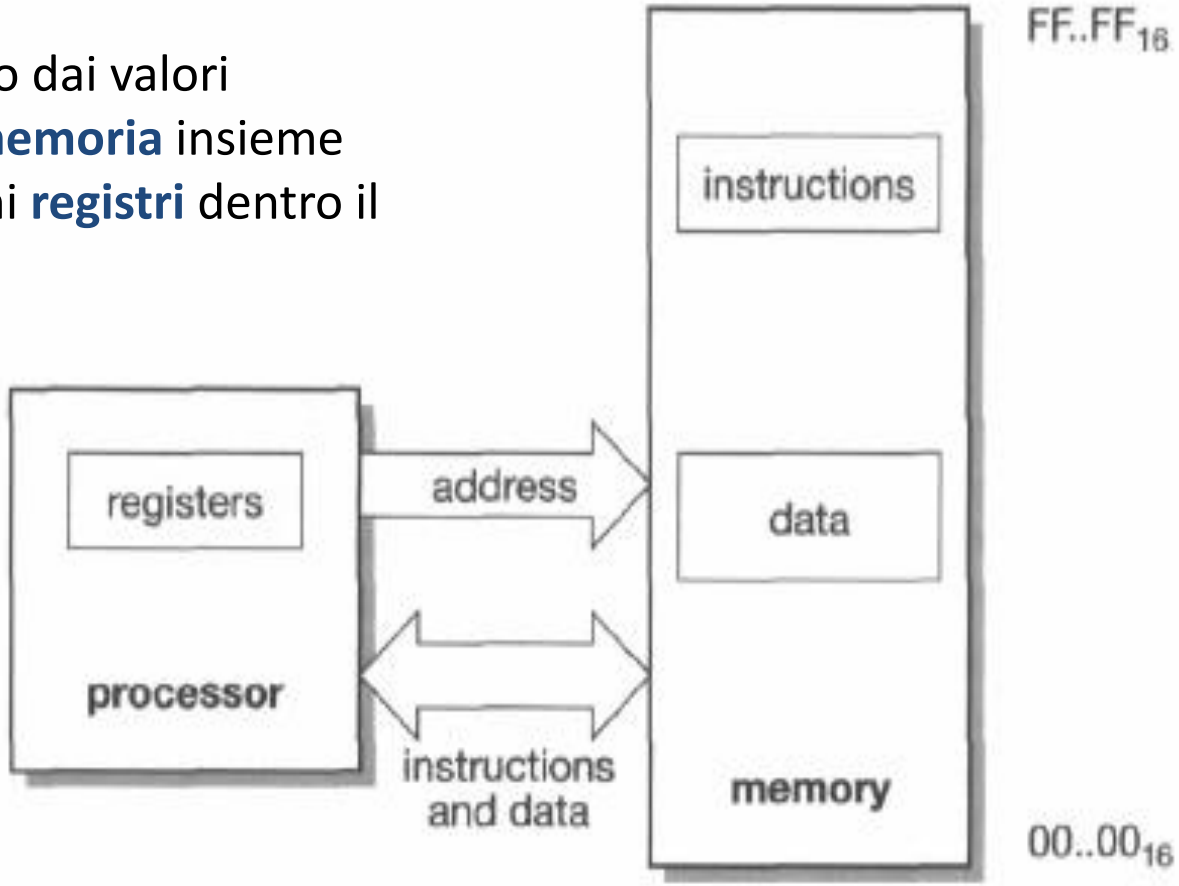
- Data-processing;
- Memory;
- Branch.

Cos'è un processore

Un processore general-purpose è un **automa** a stati finiti, che esegue **istruzioni** rilocate in una memoria.

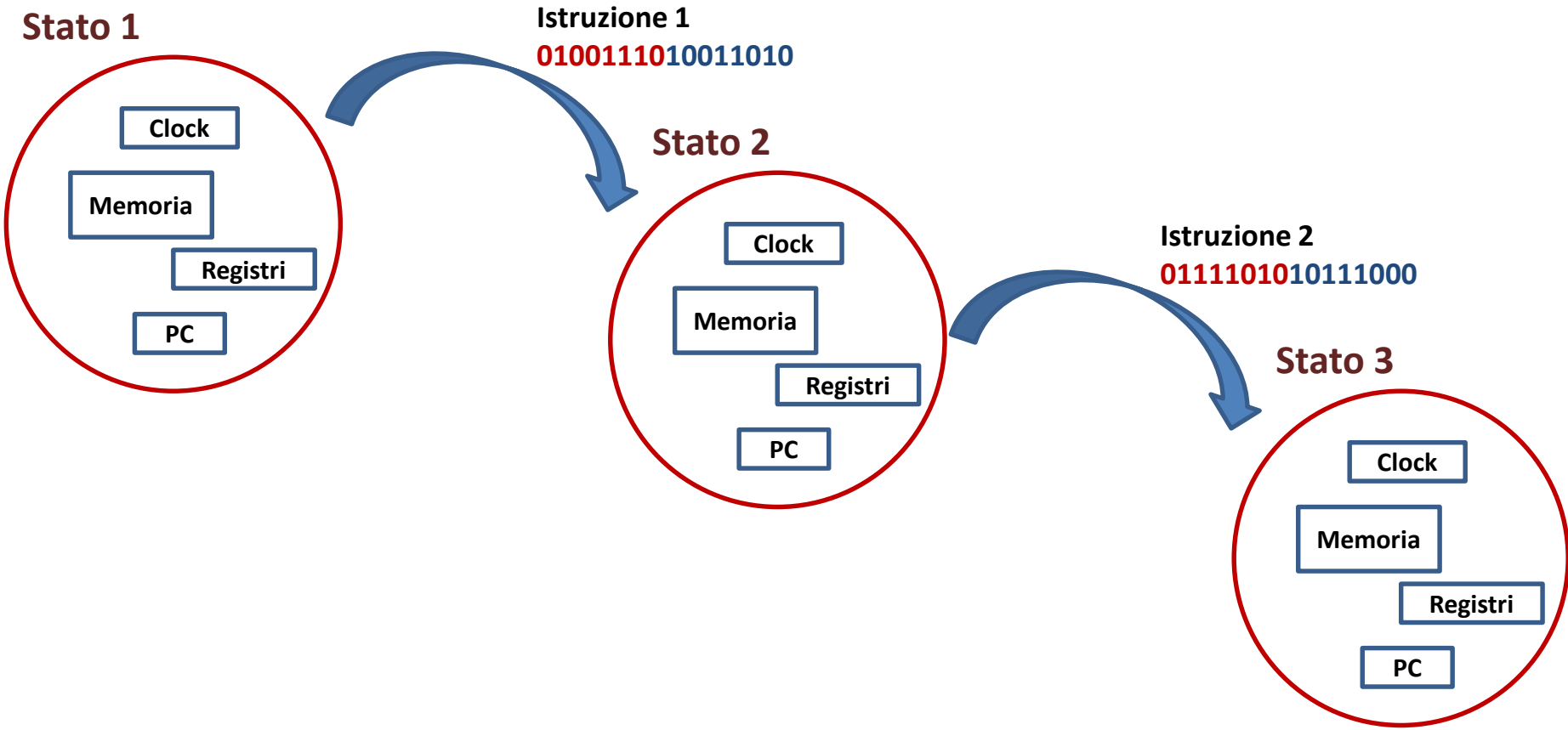
Lo **stato** del sistema è definito dai valori contenuti nelle locazioni di **memoria** insieme con i valori contenuti in alcuni **registri** dentro il processore stesso.

Ogni **istruzione** definisce in che modo lo stato deve cambiare e quale istruzione deve essere eseguita successivamente.



Microarchitetture come automi

Una microarchitettura può essere vista come un automa.



Variazioni di stato

- I registri, la memoria dati e la memoria istruzioni operano mediante logica combinatoria.
- Tutti i componenti effettuano la scrittura sul fronte alto del clock, cosicché lo stato del sistema cambia solo su un fronte del clock.
- Gli indirizzi, i dati ed il segnale di write enable devono essere impostati prima del fronte del clock e mantenuti immutati per un tempo superiore al ritardo di propagazione.
- Sia gli elementi di stato, che il microprocessore sono costituiti da logica combinatoria e da componenti sincronizzate dal clock. L'intero sistema è, quindi, sincrono e può essere visto come una complessa macchina a stati finiti o come l'insieme di macchine a stati finiti semplici, che interagiscono fra loro.

Microarchitetture a ciclo singolo

Saranno prese in considerazione tre diverse microarchitetture ARM, le quali differiscono fra loro principalmente per il modo in cui gli elementi di stato sono interconnessi.

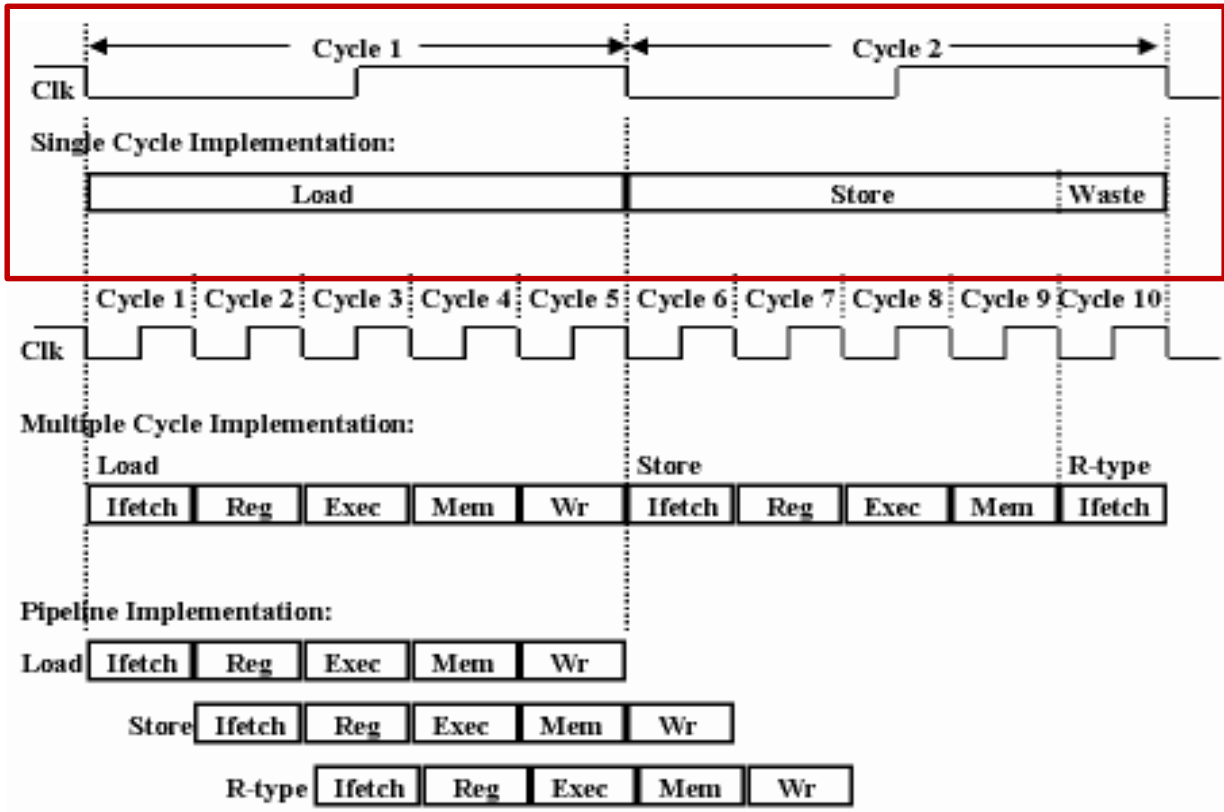
Ciclo singolo: l'intera istruzione è eseguita in un singolo ciclo.

Vantaggi:

- semplice da comprendere;
- unità di controllo molto semplice;
- non richiede stati non architetturali.

Svantaggi:

- tempo pari a quello dell'istruzione più lenta;
- memoria dati e memoria istruzioni separate;



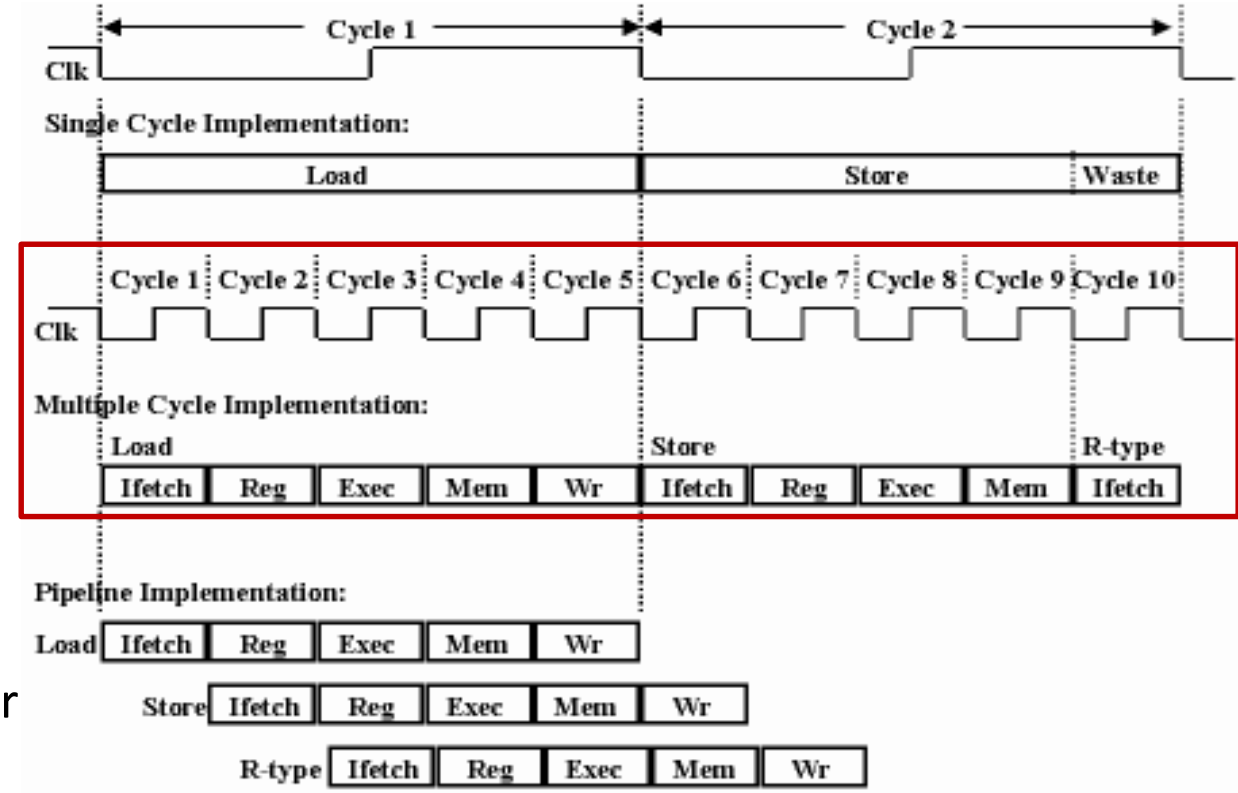
Microarchitetture a ciclo multiplo

Saranno prese in considerazione tre diverse microarchitetture ARM, le quali differiscono fra loro principalmente per il modo in cui gli elementi di stato sono interconnessi.

Ciclo multiplo: esegue una istruzione in più cicli brevi.

Vantaggi:
- riuso dei componenti;
- durata variabile delle istruzioni;
- non richiede la separazione delle memorie.

Svantaggi:
- richiede stati non architetturali;
- esegue una istruzione per volta.



Microarchitetture con pipeline

Saranno prese in considerazione tre diverse microarchitetture ARM, le quali differiscono fra loro principalmente per il modo in cui gli elementi di stato sono interconnessi.

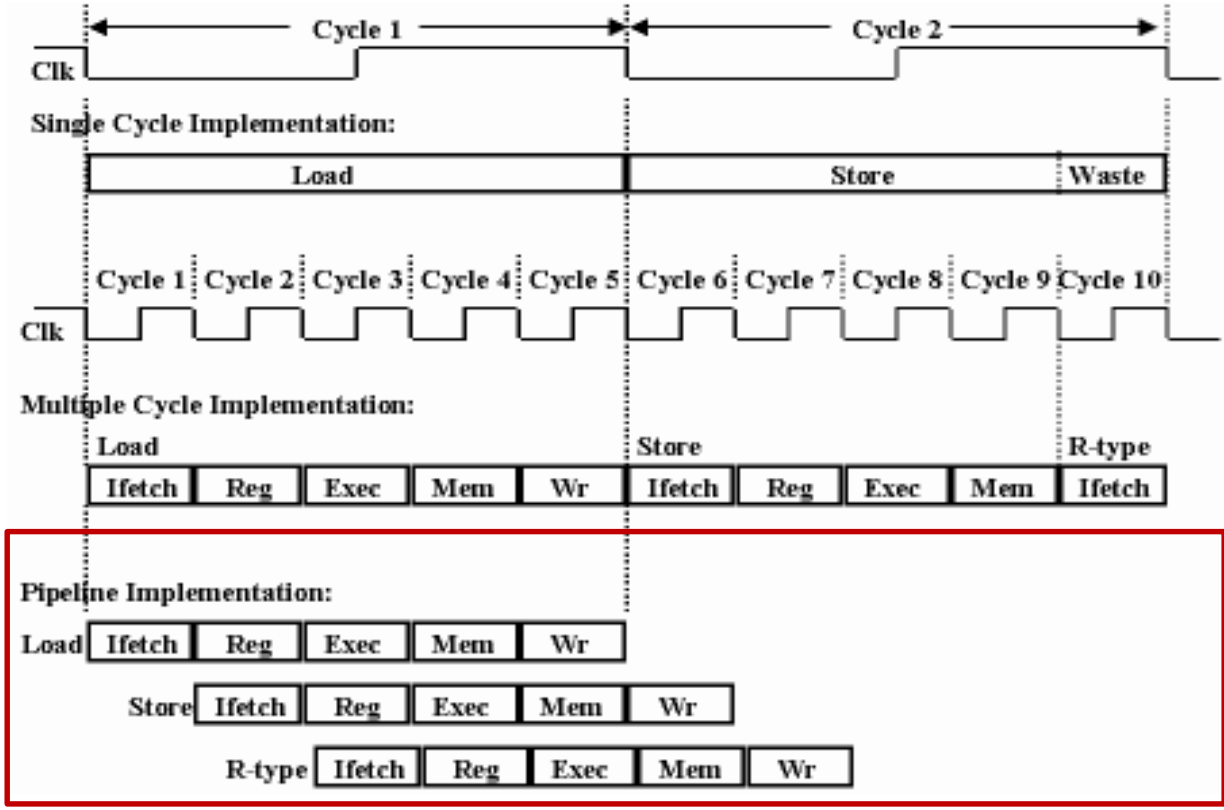
Pipelining: si applica al processore a ciclo singolo e ne migliora le performance.

Vantaggi:

- esegue più istruzioni contemporaneamente;
- si può accedere a dati e registri contemporaneamente.

Svantaggi:

- logica di controllo più complessa.
- richiede registri di pipeline.



Misura delle prestazioni

Ci sono molti modi per misurare le performance di un processore.

Una misura affidabile consiste nel valutare le prestazioni di tempo rispetto all'esecuzione di un insieme fissato di programmi, che prende il nome di

CINT2006 (Integer Component of SPEC CPU2006):

Benchmark	Language	Application Area	Brief Description
400.perlbench	C	Programming Language	Derived from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).
401.bzip2	C	Compression	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.
403.gcc	C	C Compiler	Based on gcc Version 3.2, generates code for Opteron.
429.mcf	C	Combinatorial Optimization	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
445.gobmk	C	Artificial Intelligence: Go	Plays the game of Go, a simply described but deeply complex game.
456.hmmer	C	Search Gene Sequence	Protein sequence analysis using profile hidden Markov models (profile HMMs)
458.sjeng	C	Artificial Intelligence: chess	A highly-ranked chess program that also plays several chess variants.
462.libquantum	C	Physics / Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
464.h264ref	C	Video Compression	A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2
471.omnetpp	C++	Discrete Event Simulation	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
473.astar	C++	Path-finding Algorithms	Pathfinding library for 2D maps, including the well known A* algorithm.
483.xalancbmk	C++	XML Processing	A modified version of Xalan-C++, which transforms XML documents to other document types.

Misura delle prestazioni

Ci sono molti modi per misurare le performance di un processore.

Il tempo di esecuzione è calcolato come:

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

CPI: Cycles/instruction

clock period: seconds/cycle

IPC: instructions/cycle = IPC

Misura delle prestazioni

Ci sono molti modi per misurare le performance di un processore.

Il tempo di esecuzione è calcolato come:

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

CPI: Cycles/instruction

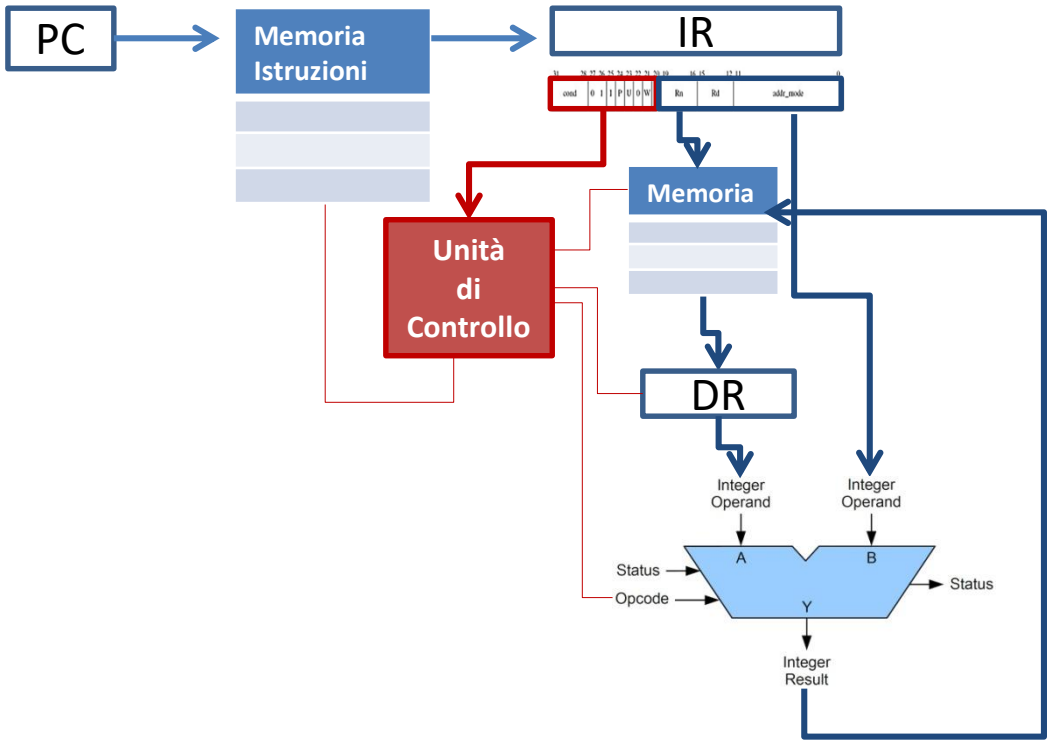
clock period: seconds/cycle

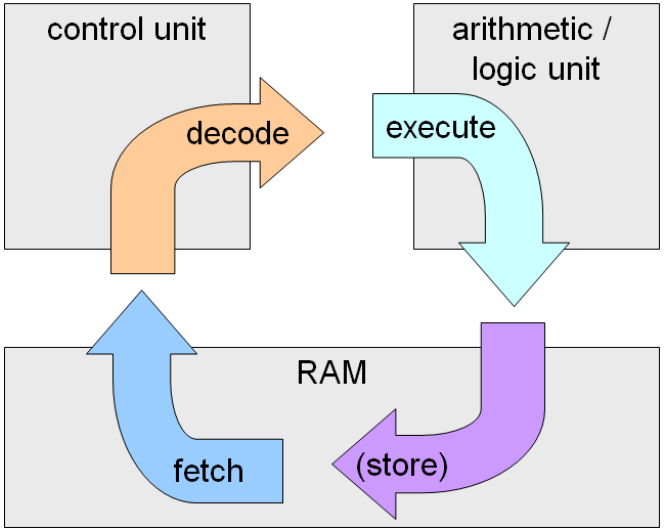
IPC: instructions/cycle = IPC

La frequenza di clock corrisponde all'inverso del clock period

Microarchitetture: schema generale

Una istruzione ha un ciclo di vita che consta di quattro fasi principali





Progettazione

Una microarchitettura consta di due componenti che interagiscono fra loro:

- il **datapath:**
  - determina il flusso dei dati durante l'esecuzione di una istruzione
  - opera su parole dati e contiene strutture quali le memorie, i registri, l'ALU, e i multiplexer
  - considereremo un datapath a 32 bit.
- l'**unità di controllo:**
  - riceve l'istruzione corrente dal datapath e dice al datapath come eseguire tale istruzione.
  - Produce i valori di selezione dei multilpexer, i segnali di abilitazione alla scrittura dei registri e della memoria.

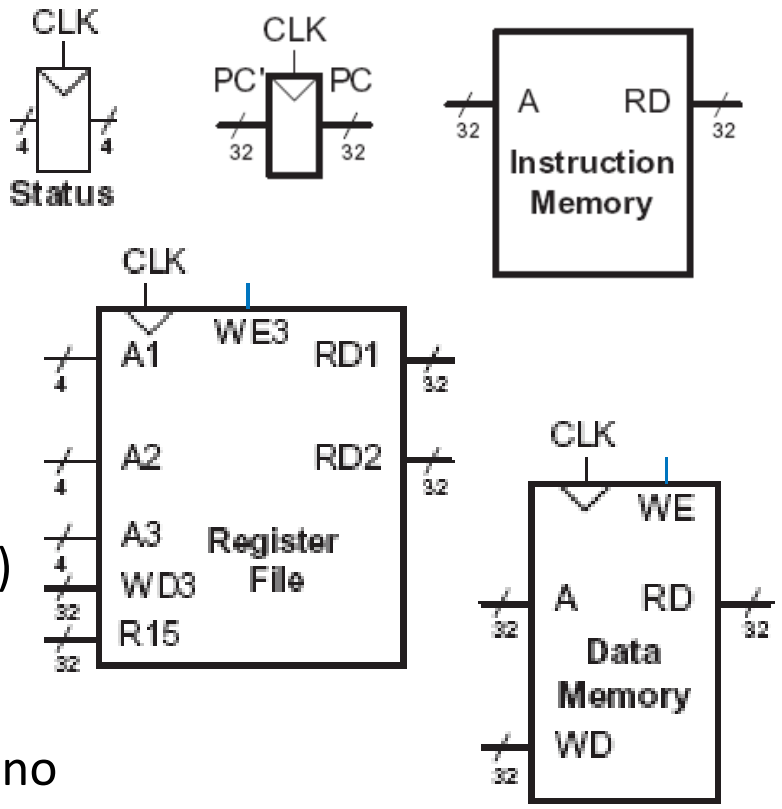
Procederemo in modo incrementale: Elementi di stato (program counter, registri, e registro di stato). Logica combinatoria fra gli elementi di stato. Gestione della memoria.

Progettazione

Il miglior modo di procedere nel processo di progettazione consiste nel considerare prima gli elementi di stato (program counter, registri, registro di stato) e aggiungere successivamente la logica combinatoria.

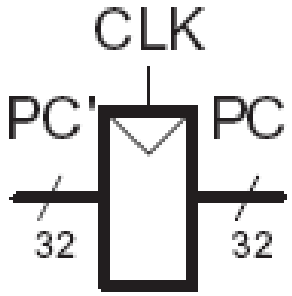
Suddivideremo la memoria in cinque elementi di stato:

- il program counter
- i registri (register file)
- il registro di stato (status register)
- la memoria istruzioni (instruction memory)
- la memoria dati (data memory).

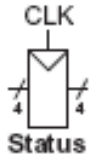


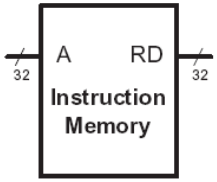
La memoria per le istruzioni e quella per i dati sono separate per il solo scopo di facilitare la comprensione.

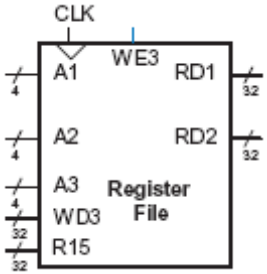
Program counter

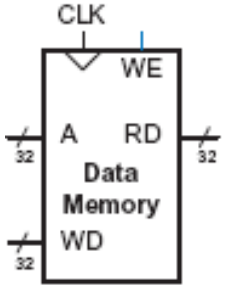


Sebbene il PC sia logicamente parte del file registro, esso viene letto e scritto ad ogni ciclo indipendentemente dagli altri registri ed è quindi implementato come un registro autonomo 32-bit. La sua uscita, PC, indica l'indirizzo dell'istruzione corrente. Il suo ingresso, PC', indica l'indirizzo della successiva istruzione.

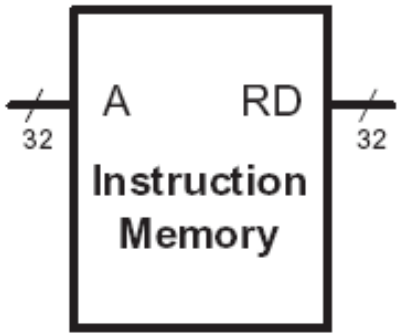








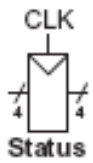
Istruction Memory

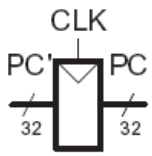


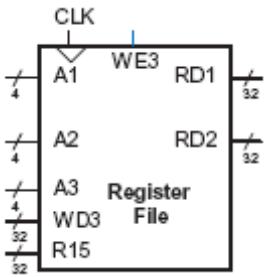
essa ha una singola porta di sola lettura (*semplificazione*).

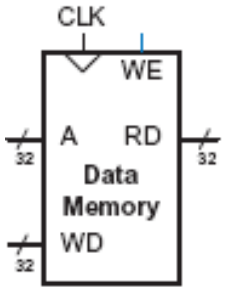
A indica l'indirizzo di una istruzione che verrà riportata (*letta*) nel registro **RD**.

Considerando che in genere le memorie sono byte addressable ed essendo sia A che RD a 32 bit, qual'è il numero massimo di istruzioni che una architettura 32 bit supporta?

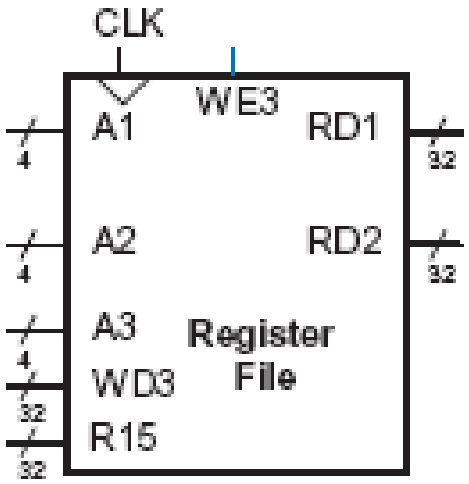


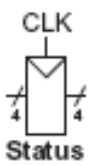


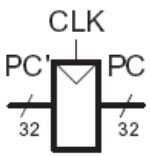


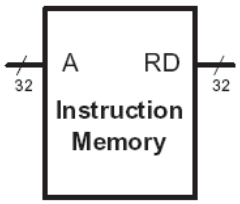


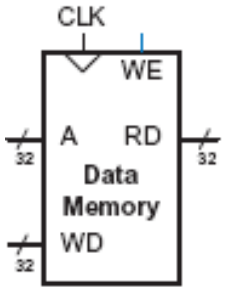
File register











consiste di 16 registri (**R0,…,R15**) a 32 bit

Dal punto di vista logico i registri non sono equivalenti, svolgono al livello architetturale funzioni diverse. In particolare:

R13: stack pointer

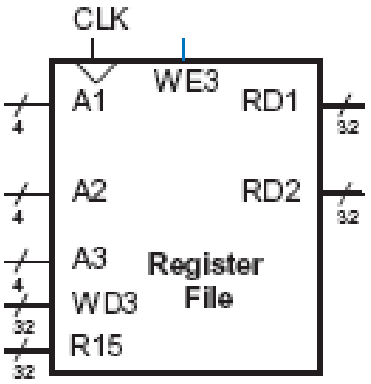
R14: link register

R15: proveniente dal program counter

File register

Esso ha due porte per la lettura **A1** e **A2** ed una per la scrittura **A3**.

Ciascuna porta di lettura ha un input di 4 bit (2^4=16), che specificano uno dei registri come operando, che viene letto nei registri **RD1** o **RD2**, rispettivamente.

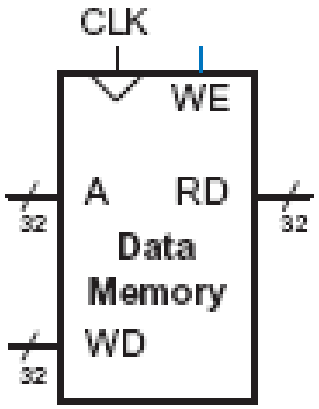


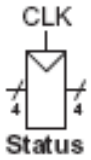
La porta di scrittura ha:

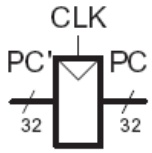
- un indirizzo di 4 bit **A3** (dove scrivere);
- un elemento di 32 bit **WD3** (cosa scrivere);
- un segnale di Write Enable (**WE3**).

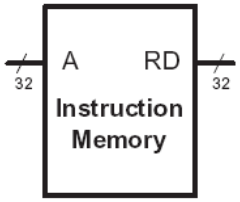
Se **WE** è 1, il dato **WD3** è scritto nel registro specificato da **A3**.

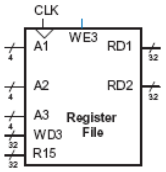
Data Memory





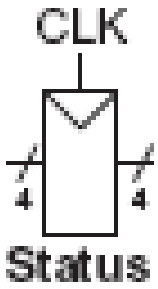




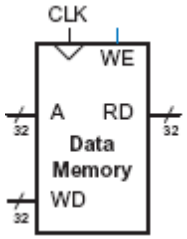


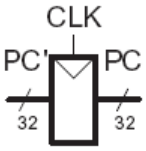
Il **Data Memory**: ha una singola porta di lettura/scrittura.
Quando il segnale **WE** (Write Enable) è attivo, essa scrive il dato **WD** nella cella puntata dall'indirizzo **A** durante il fronte alto del clock.
Quando il segnale **WE** (Write Enable) è 0, essa legge i dati nella cella puntata dall'indirizzo **A** durante il fronte alto del clock e li pone nel registro **RD**.

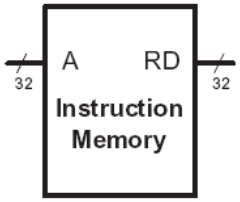
Status

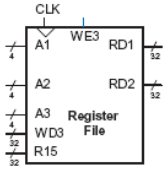


Memorizza i flags **N**,**Z**,**C**,**V** fornite dall'ALU al fine di eseguire istruzioni condizionate









Flag	Name	Description
N	**N**egative	Instruction result is negative
Z	**Z**ero	Instruction results in zero
C	**C**arry	Instruction causes an unsigned carry out
V	o**V**erflow	Instruction causes an overflow

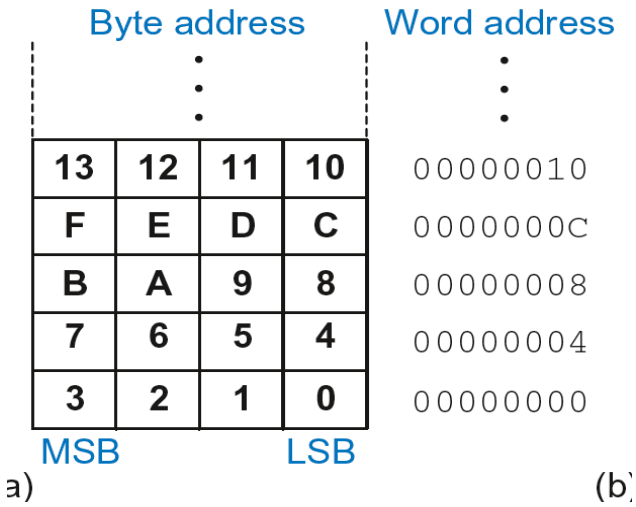
Istruzioni di base

Al fine di facilitare la comprensione dell'architettura di un processore ARM, considereremo un limitato **set** di istruzioni:

- **Istruzioni di Data-processing:**
  - `ADD, SUB, AND, ORR`
  - Con registri e modalità di indirizzamento diretto, ma senza shifts
- **Istruzioni di memoria:**
  - `LDR, STR`
  - with **positive immediate offset**
- **Istruzioni di salto:**
  - `B`

Indirizzi di memoria

- Ad ogni **byte** corrisponde un unico indirizzo
- Una parola è di 32-bit = 4 bytes, quindi gli indirizzi delle parole incrementano di 4 in 4



- Indirizzo delle seconda parola = $2 \times 4 = 8$
- Indirizzo della decima parola = $10 \times 4 = 40$

Istruzioni di memoria: load

Una istruzione che legge in memoria è detta ***load***

- **Mnemonico:** `LDR` (*load register*)
- Vi sono differenti modi per eseguire un load, ad esempio:

```
LDR R0, [R1, #12]
```

indirizzo:

- *base address* + *offset*
- indirizzo = (R1 + 12)

risultato:

- Il dato memorizzato all'indirizzo (R1 + 12) viene *caricato* nel registro R0

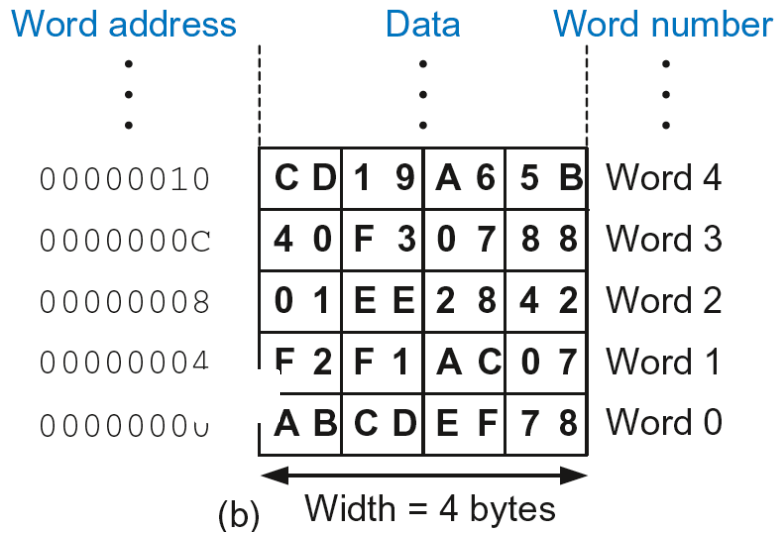
Istruzioni di memoria: load

Esempio: Carica nel registro R3 la parola all'indirizzo 8

```
MOV R2, #0
LDR R3, [R2, #8]
```

indirizzo = (R2 + 8) = 8

R3 = 0x01EE2842 (dopo il load)



Istruzioni di memoria: load

Vi sono anche altre segnature di LDR, ad esempio:

```
LDR R0, [R1, R2]
```

R0 ← mem32[R1 + R2]

```
LDR R0, [R1, R2, LSL #2]
```

R0 ← mem32[R1 + (R2*4)]

Istruzioni di memoria: store

Una istruzione che scrive in memoria è detta ***store***

- **Mnemonico:** `STR` (*store register*)
- Semantica speculare a LDR:

```
STR R0, [R1, #12]
```

mem32[R1 + 12] ← R0

- Segnature ananoghe

```
STR R1, [R0, R2]
```

mem32[R0 + R2] ← R1

```
STR R0, [R1, R2, LSL #1]
```

mem32[R1 + (R2*2)] ← R0

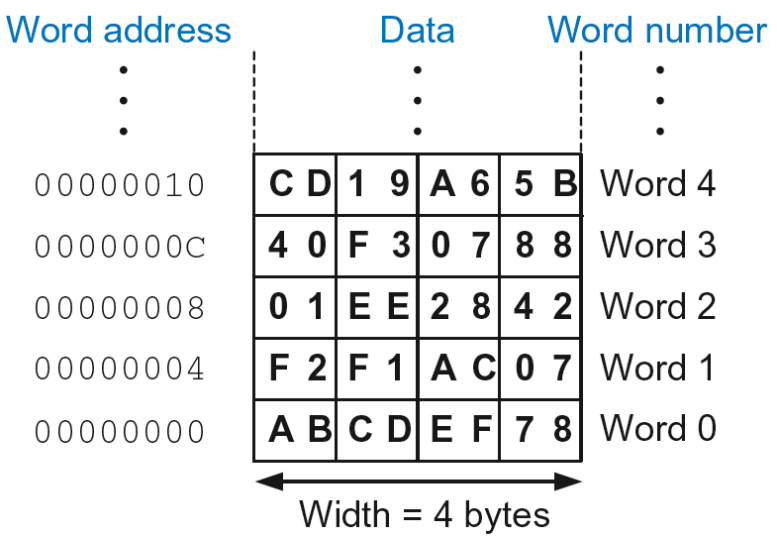
Istruzioni di memoria: store

Esempio: Memorizza il valore di R7 nella parola di memoria 21.

Indirizzo di memoria= 4 x 21 = 84 = 0x54

```
MOV R5, #0
STR R7, [R5, #0x54]
```

L'offset può essere scritto in decimale o esadecimale

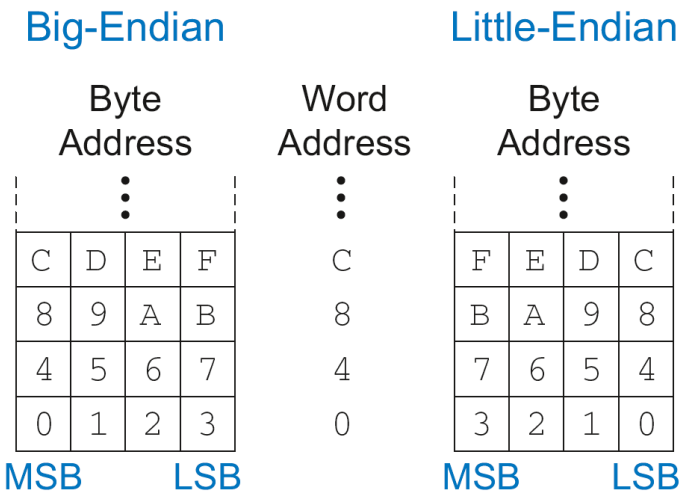


Varianti: LDRB, STRB

LDRB/STRB eseguono rispettivamente il load e lo store di un singolo byte

Come sono indirizzati i byte all'interno di una parola?

Little-endian: la numerazione dei byte inizia dal byte meno significativo
Big-endian: la numerazione dei byte inizia dal byte più significativo

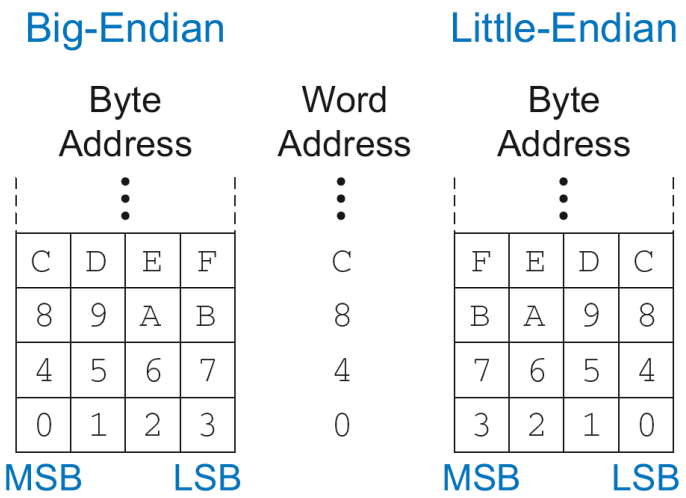


Varianti: LDRB, STRB

`LDRB/STRB` eseguono rispettivamente il load e lo store di un singolo byte

Come sono indirizzati i byte all'interno di una parola?

Little-endian: la numerazione dei byte inizia dal byte meno significativo
Big-endian: la numerazione dei byte inizia dal byte più significativo



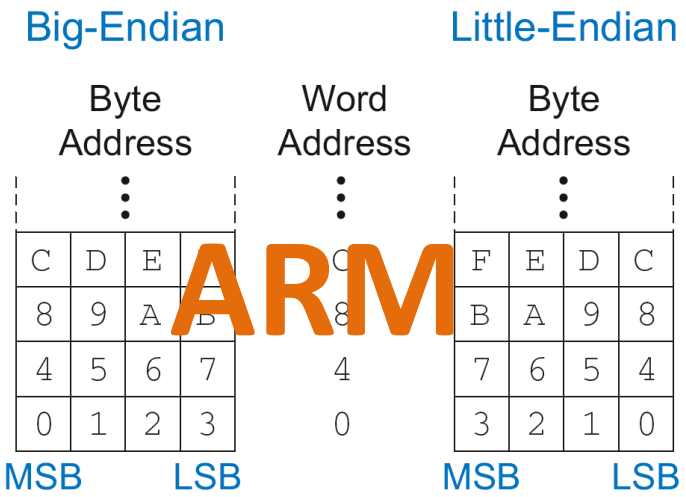
IBM PowerPC **Intel x86**

Varianti: LDRB, STRB

`LDRB/STRB` eseguono rispettivamente il load e lo store di un singolo byte

Come sono indirizzati i byte all'interno di una parola?

Little-endian: la numerazione dei byte inizia dal byte meno significativo
Big-endian: la numerazione dei byte inizia dal byte più significativo

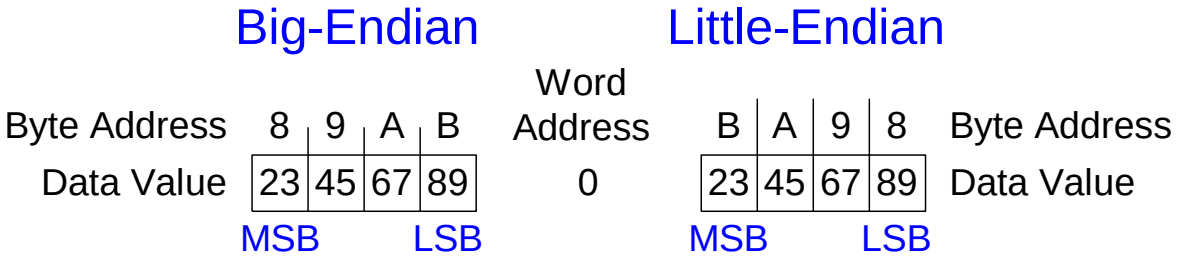


Varianti: LDRB, STRB

Esempio: si assuma che R2 *vale* 0 (0x00000000) e che si esegua l'istruzione

```
LDRB R7, [R2, #1]
```

Quale sarà il valore di di R7 in un sistema little-endian/ big-endian?



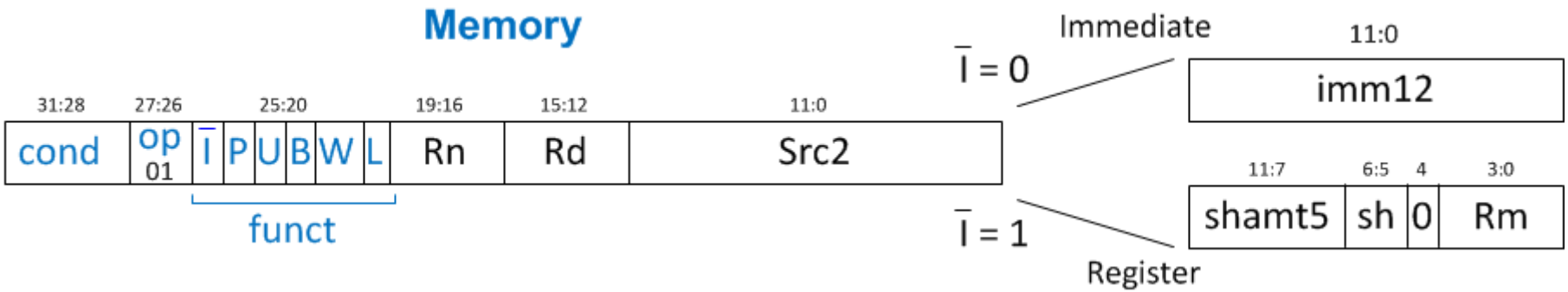
Big-endian:
0x00000045

Little-endian:
0x00000067

Codifiche istruzioni di memoria

`LDR`, `STR`, `LDRB`, `STRB`

- ***op*** = 01_2
- ***Rn*** = registro *base address*
- ***Rd*** = destinazione (load), sorgente (store)
- ***Src2*** = offset: registro (*shiftato* opzionalmente) o immediate
- ***funct*** = 6 bit di controllo



Indicizzazione

ARM fornisce tre tipi di indicizzazione:

Offset:
L'indirizzo è calcolato sommando o sottraendo un offset al contenuto del registro di base.

Preindex
L'indirizzo viene calcolato come nel caso dell'offset e viene scritto nel registro di base;

Postindex
L'indirizzo corrisponde al contenuto del registro di base;
L'offset viene aggiunto o sottratto al contenuto del registro di base e riscritto nel registro di base

Table 6.4 ARM indexing modes

Mode	ARM Assembly	Address	Base Register
Offset	`LDR R0, [R1, R2]`	R1 + R2	Unchanged
Pre-index	`LDR R0, [R1, R2]!`	R1 + R2	R1 = R1 + R2
Post-index	`LDR R0, [R1], R2`	R1	R1 = R1 + R2

Indicizzazione

ARM fornisce tre tipi di indicizzazione:

Offset:
L'indirizzo è calcolato sommando o sottraendo un offset al contenuto del registro di base.

Preindex
L'indirizzo viene calcolato come nel caso dell'offset e viene scritto nel registro di base;

Postindex
L'indirizzo corrisponde al contenuto del registro di base;
L'offset viene aggiunto o sottratto al contenuto del registro di base e riscritto nel registro di base

Esempi

Offset:

```
LDR R1, [R2, #4]          ; R1 = mem[R2+4]
```

Preindex:

```
LDR R3, [R5, #16]!        ; R3 = mem[R5+16]
                          ; R5 = R5 + 16
```

Postindex:

```
LDR R8, [R1], #8          ; R8 = mem[R1]
                          ; R1 = R1 + 8
```

Campi di funct

Type of Operation

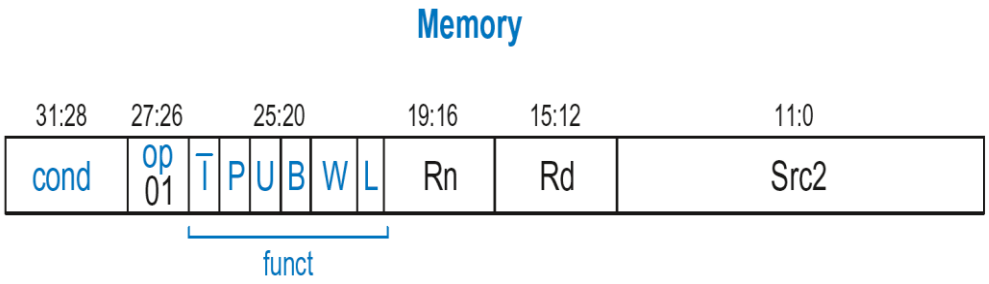
L	*B*	Instruction
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

Indexing Mode

P	*W*	Indexing Mode
0	1	Not supported
0	0	Postindex
1	0	Offset
1	1	Preindex

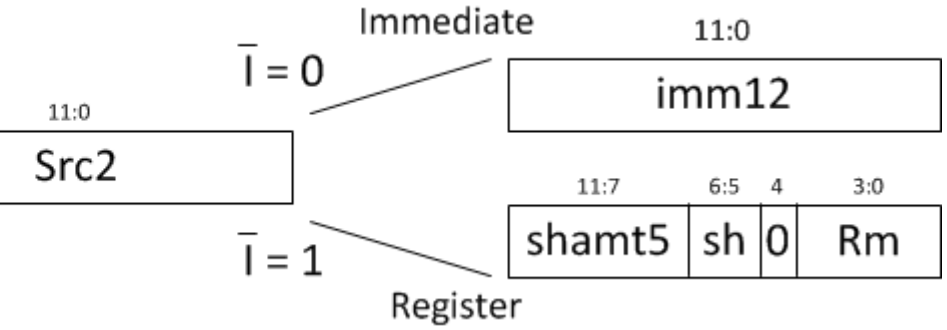
Add/Subtract Immediate/Register Offset

Value	$\overline{I}$	*U*
0	**Immediate** offset in *Src2*	**Subtract** offset from base
1	**Register** offset in *Src2*	**Add** offset to base



Campo sh

Sh codifica il tipo di shift (i.e., >>, <<, >>>, ROR)



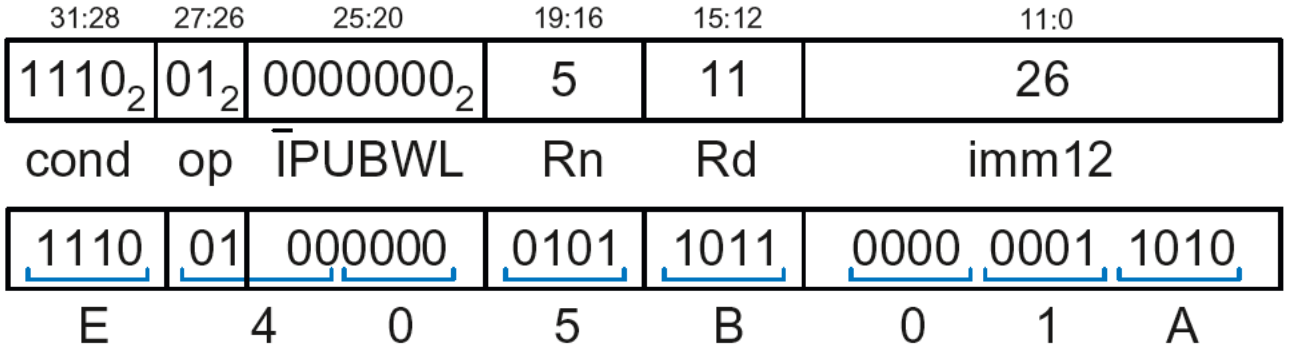
Shift Type	*sh*
LSL	00_2
LSR	01_2
ASR	10_2
ROR	11_2

Esempi

```
STR R11, [R5], #-26
```

- **Operation:** mem[R5] <= R11; R5 = R5 - 26
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 01_2 (1) for memory instruction
- ***funct*** = $0\bar{0}000000_2$ (0)
- ***I*** = 0 (immediate offset), ***P*** = 0 (postindex),
- ***U*** = 0 (subtract), ***B*** = 0 (store word), ***W*** = 0 (postindex),
- ***L*** = 0 (store)
- ***Rd*** = 11, ***Rn*** = 5, ***imm12*** = 26

Field Values

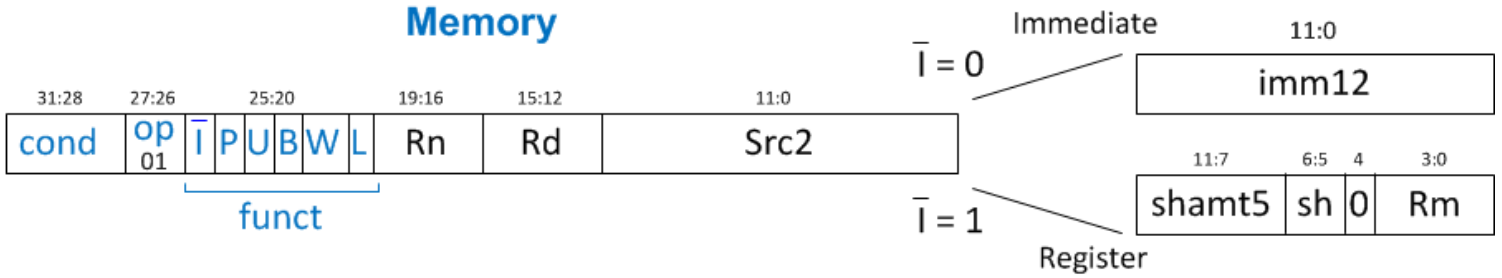


Esempi

```
LDR R3, [R4, R5]
```

- **Operation:** R3 <= mem[R4 + R5]
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 01_2 (1) for memory instruction
- ***funct*** = $1\bar{I}1001_2$ (57)
- ***I*** = 1 (register offset), ***P*** = 1 (offset indexing),
- ***U*** = 1 (add), ***B*** = 0 (load **word**), ***W*** = 0 (offset indexing),
- ***L*** = 1 (load)
- ***Rd*** = 3, ***Rn*** = 4, ***Rm*** = 5 (***shamt5*** = 0, ***sh*** = 00)

1110 01 111001 0100 0011 00000 00 0 0101 = **0xE7943005**

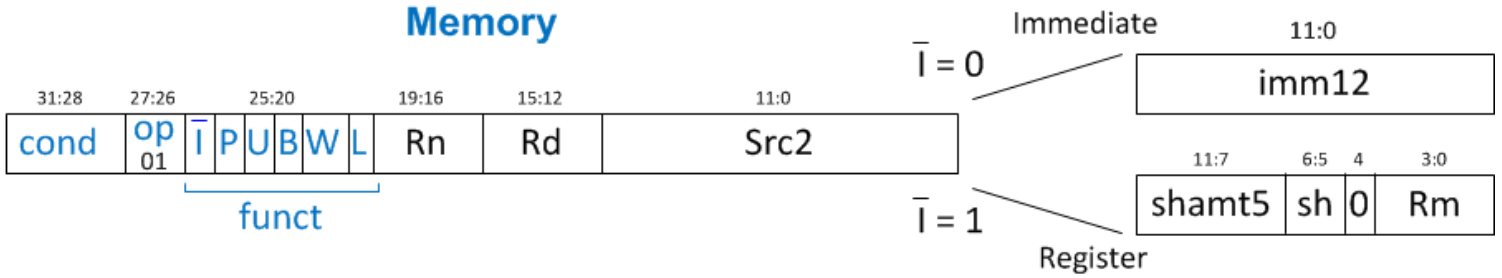


Esempi

```
STR R9, [R1, R3, LSL #2]
```

- **Operation:** mem[R1 + (R3 << 2)] <= R9
- ***cond*** = 1110_2 (14) for unconditional execution
- ***op*** = 01_2 (1) for memory instruction
- ***funct*** = 111000_2 (0)
- ***I*** = 1 (register offset), ***P*** = 1 (offset indexing),
- ***U*** = 1 (add), ***B*** = 0 (store **word**), ***W*** = 0 (offset indexing),
- ***L*** = 0 (store)
- ***Rd*** = 9, ***Rn*** = 1, ***Rm*** = 3, ***shamt*** = 2, ***sh*** = 00_2 (LSL)

1110 01 111000 0001 1001 00010 00 0 0011 = **0xE7819103**



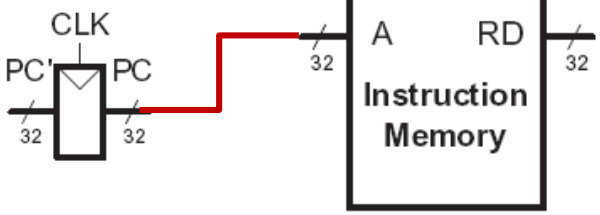
Datapath LDR

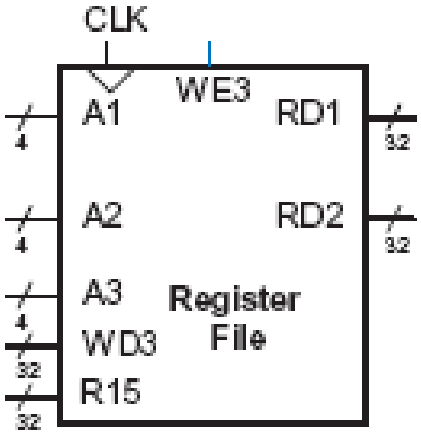
`LDR Rd, [Rn, imm12]`

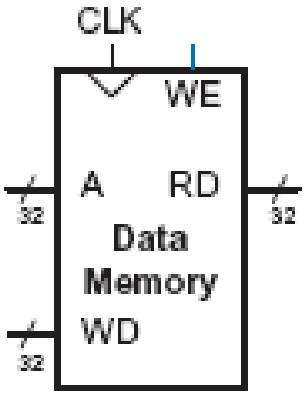
Il **PC** contiene l'indirizzo dell'istruzione da eseguire.

Il primo passo è quello di leggere questa istruzione dalla memoria istruzioni, per cui il PC viene collegato all'indirizzo di ingresso della memoria di istruzioni.

L'istruzione a 32 bit viene letta ed è rappresentata dall'etichetta **Instr**.





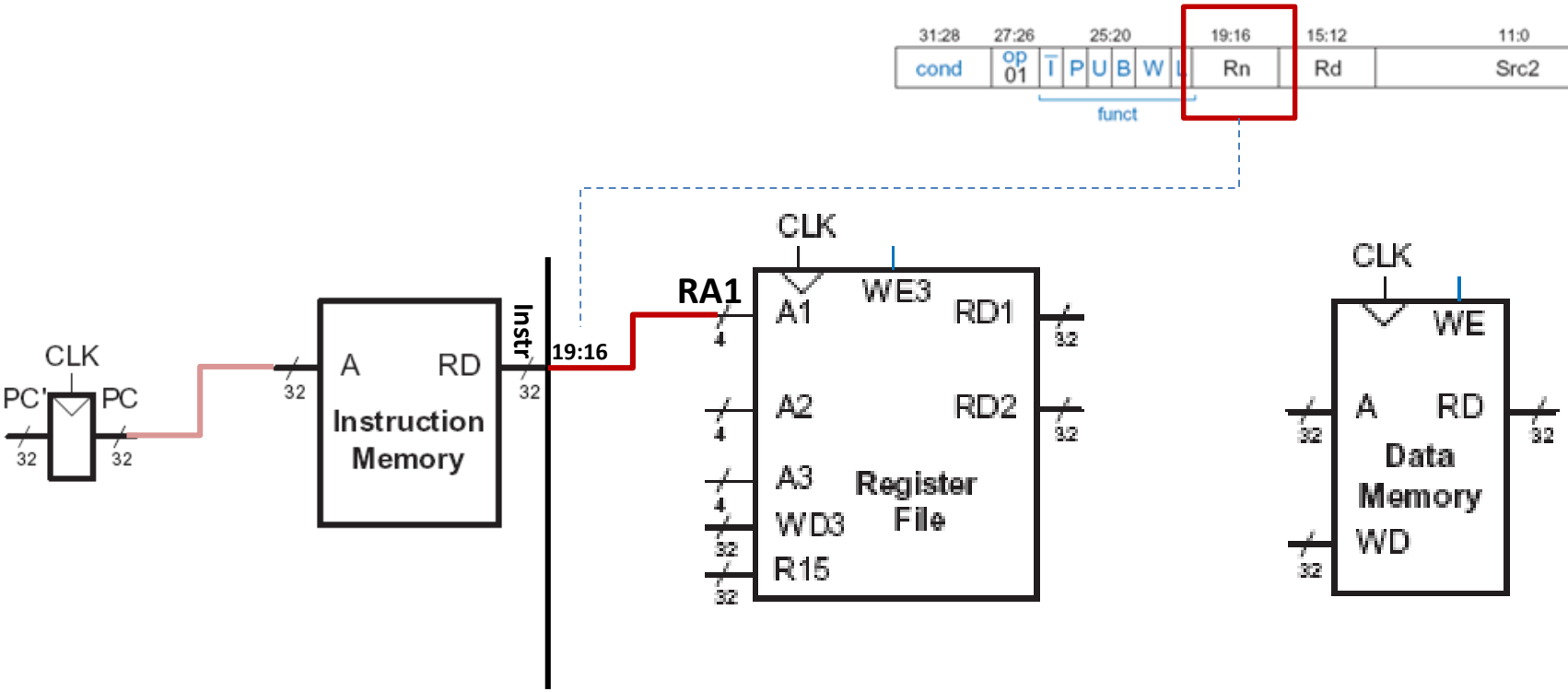


Datapath LDR

Il passo successivo è quello di leggere il registro sorgente contenente l'indirizzo di base. Questo registro è specificato nel campo **Rn** dell'istruzione, $\mathbf{Instr}_{19:16}$

Questi bit vengono collegati all'ingresso indirizzo di una delle porte del file register (**A1**).

Il register file legge il valore di registro in **RD1**.

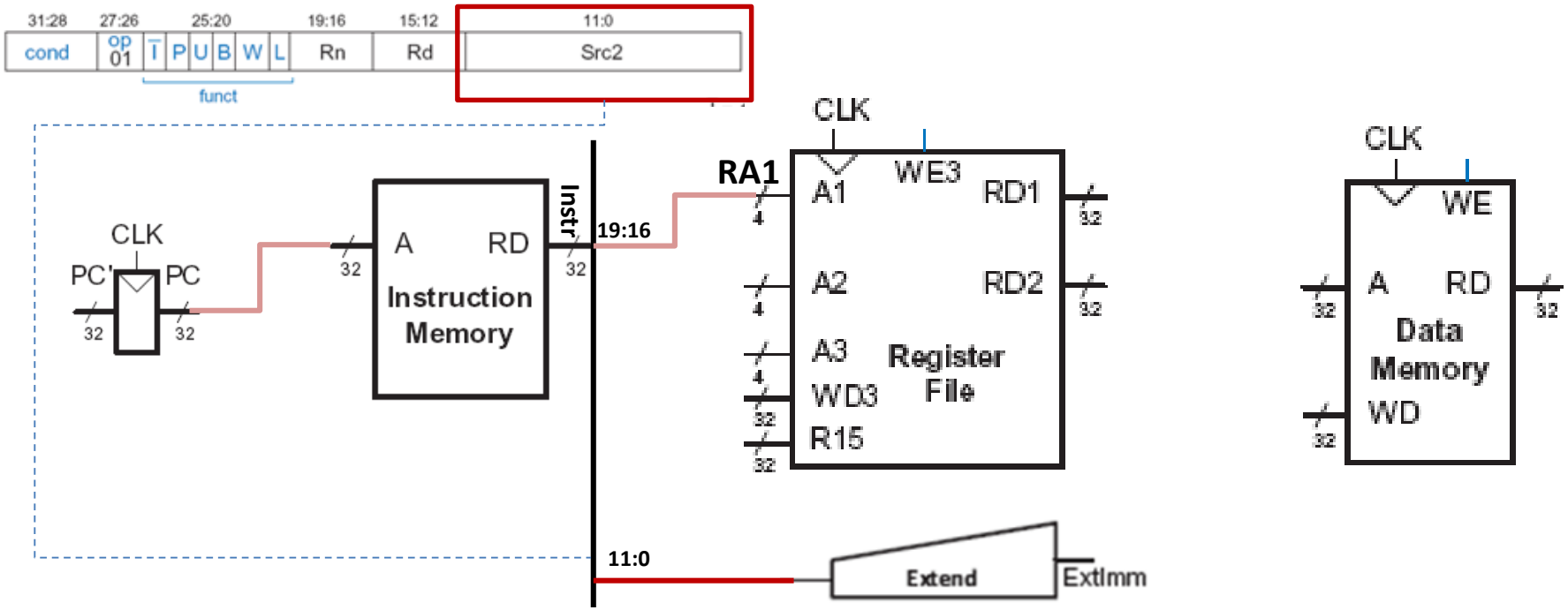


Datapath LDR

L'istruzione **LDR** richiede anche un **offset**, il quale è memorizzato nell'istruzione stessa e corrisponde ai bit $\mathbf{Instr}_{11:0}$.

L'offset è un valore senza segno, quindi deve essere esteso a 32 bit.

Il valore a 32 bit (**ExtImm**) è tale che $\mathbf{ExtImm}_{31:12} = 0$ e $\mathbf{ExtImm}_{11:0} = \mathbf{Instr}_{11:0}$.

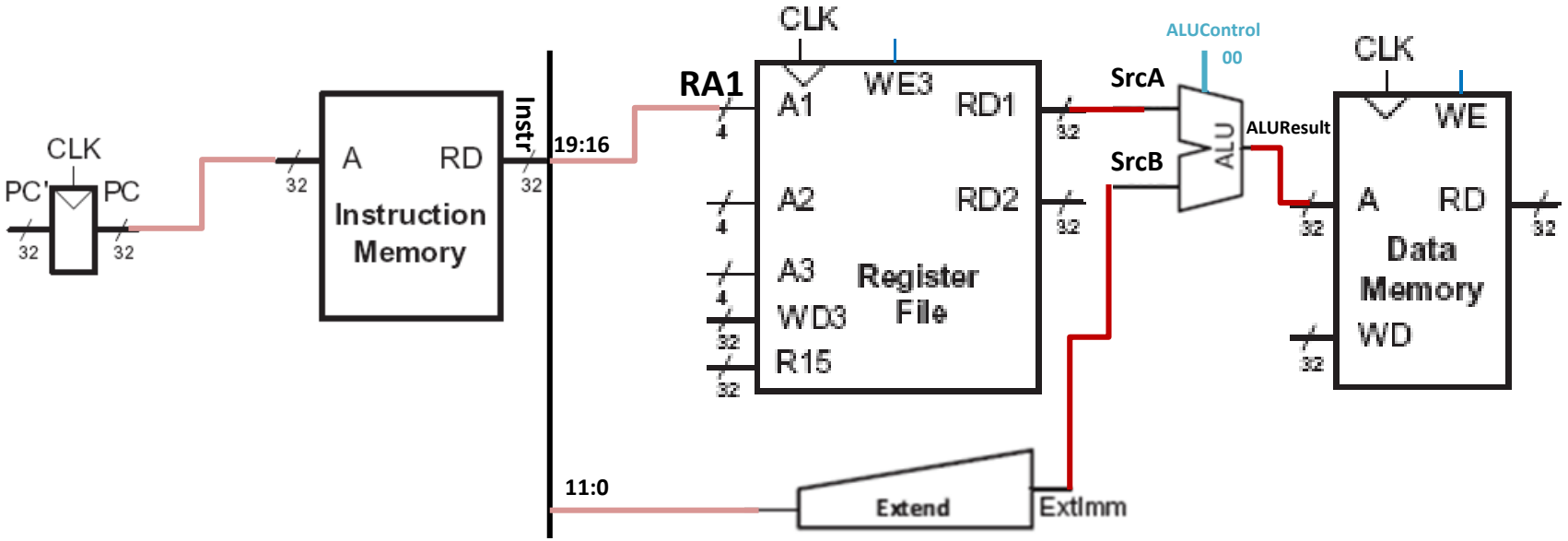


Datapath LDR

Il processore deve aggiungere l'**indirizzo di base** all'offset per trovare l'indirizzo di memoria a cui leggere. La somma è effettuata per mezzo di una **ALU**.

La **ALU** riceve due operandi (**srcA** e **srcB**). **srcA** proviene dal register file, mentre e **srcB** in questo esempio proviene da **ExtImm**. Inoltre, il segnale a 2-bit **ALUControl** specifica l'operazione (una somma è indicata con 00, se il valore del bit U è uguale a 0 allora ALUControl sarebbe 01).

La **ALU** genera un valore a 32 bit **ALUResult**, che viene inviato alla memoria dati come indirizzo di lettura.

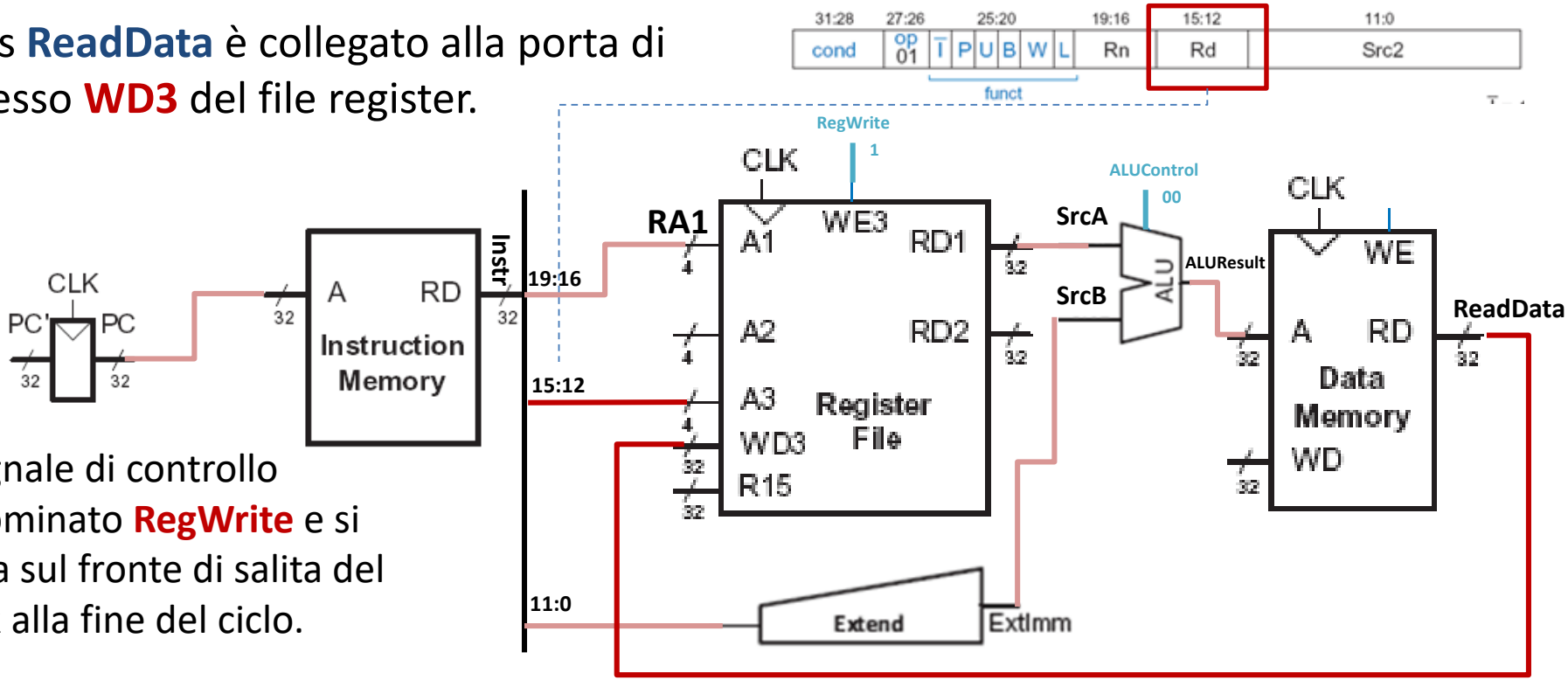


Datapath LDR

I dati vengono letti dalla memoria dati sul bus **ReadData** e poi sono scritti nel registro destinazione alla fine del ciclo.

il registro di destinazione per l'istruzione **LDR** è specificato nel campo **Rd**, **Instr$_{15:12}$**, che è collegato all'indirizzo di ingresso **A3**.

Il bus **ReadData** è collegato alla porta di ingresso **WD3** del file register.



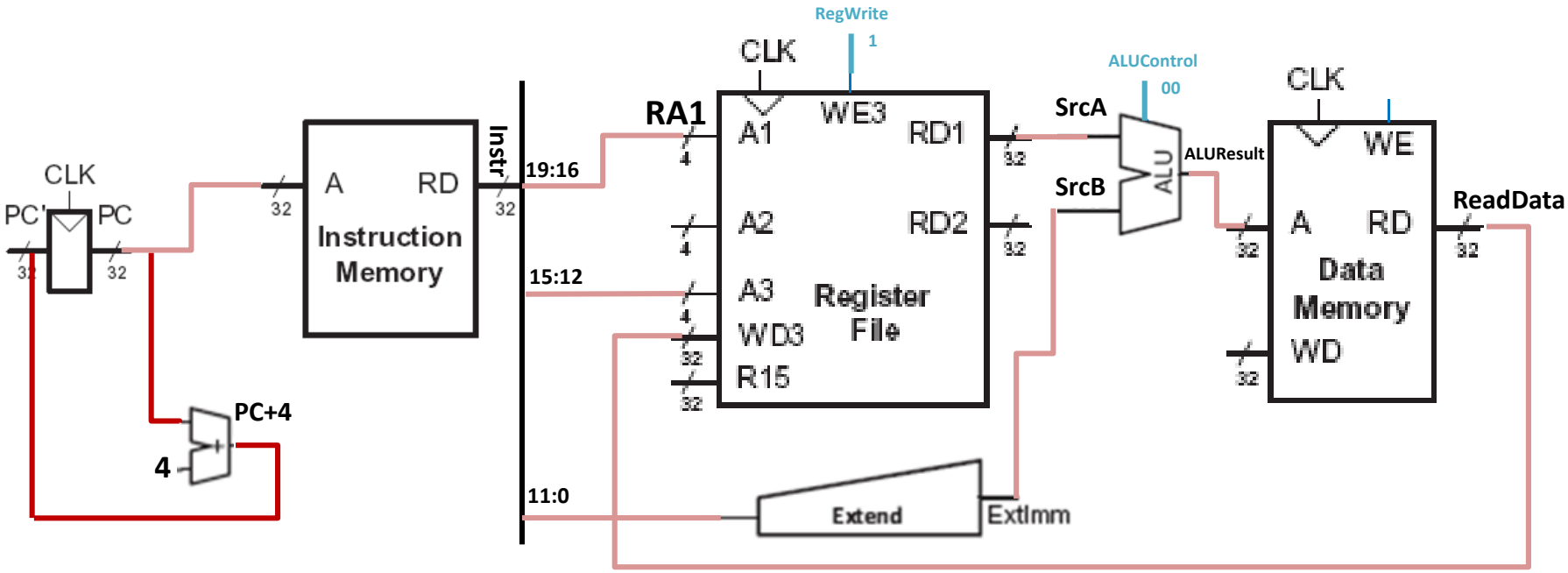
Il segnale di controllo denominato **RegWrite** e si attiva sul fronte di salita del clock alla fine del ciclo.

Datapath LDR

Contemporaneamente il processore deve calcolare l'indirizzo della successiva istruzione **PC '**.

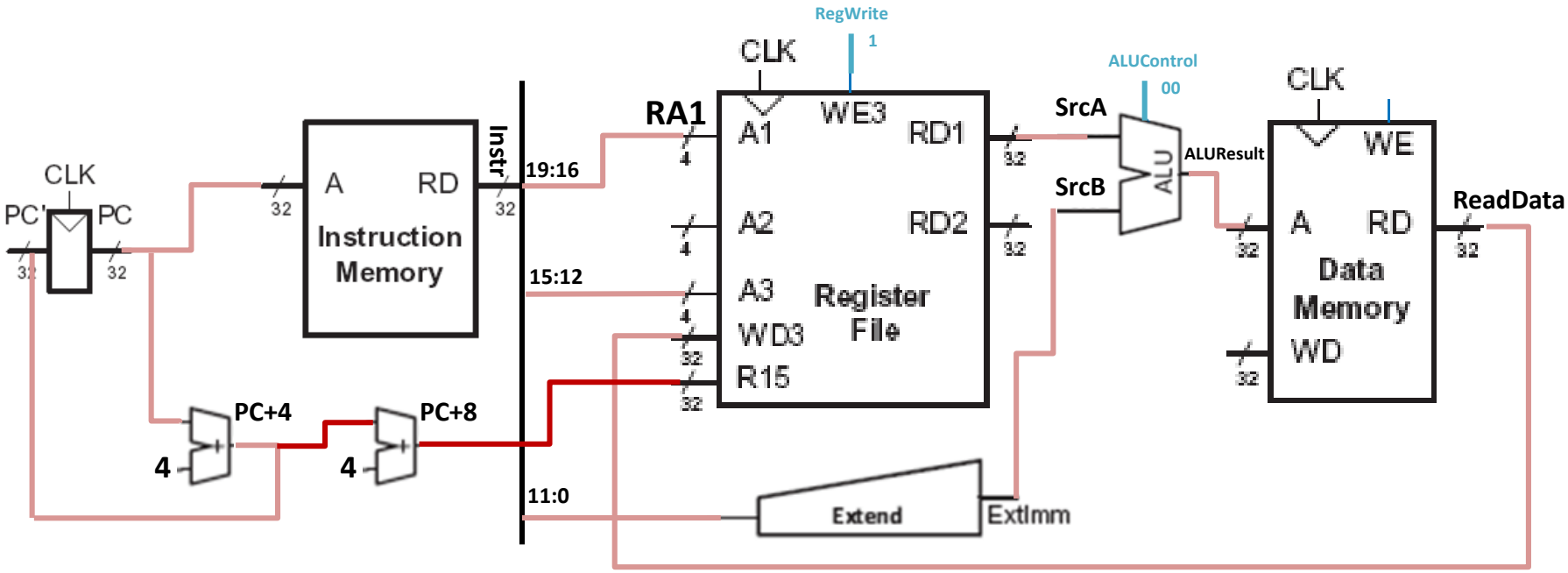
Le istruzioni sono a 32 bit (4 byte), quindi l'istruzione successiva è a **PC + 4**.

Si utilizza un **sommatore** per incrementare il **PC** di 4. Il nuovo indirizzo viene scritto nel contatore di programma sul successivo fronte di salita del clock.



Datapath LDR

Nelle architetture ARM il registro **R15** contiene il valore **PC+8**, per cui è necessario un ulteriore **sommatore** (+4), la cui uscita sia collegata all'ingresso di **R15**.

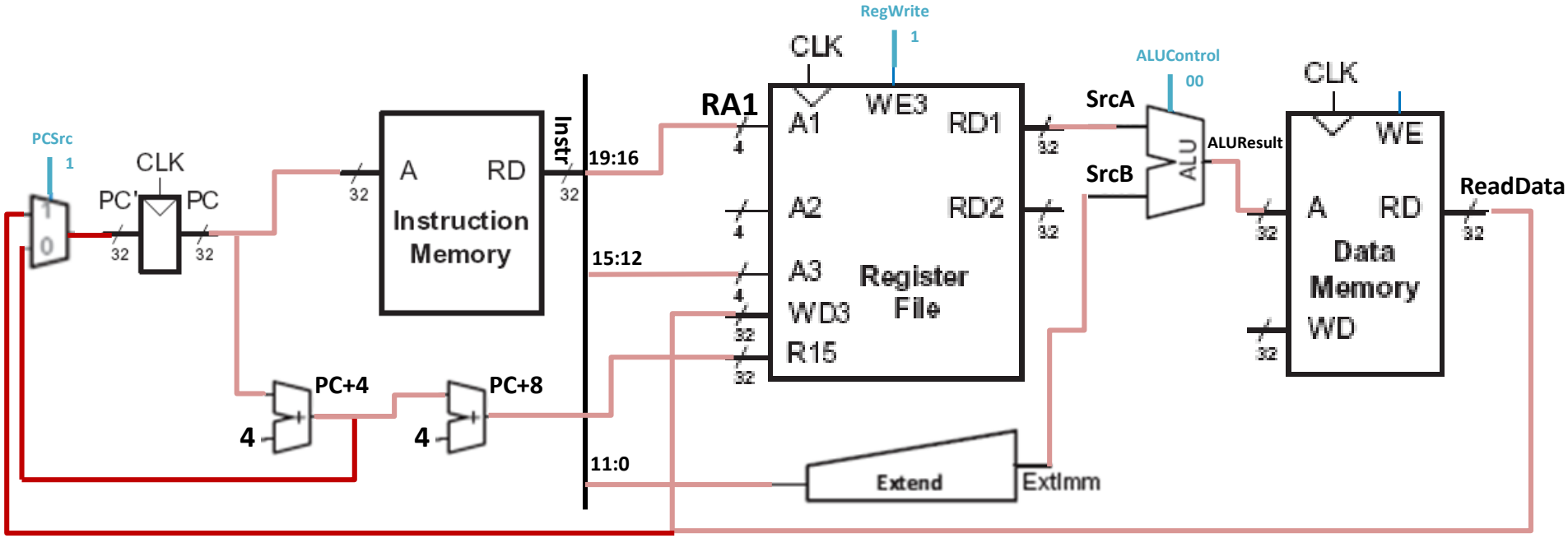


Datapath LDR

Infine, trattandosi di uno stored program paradigm, l'istruzione successiva potrebbe essere letta anche dalla memoria e quindi corrispondere al contenuto di **ReadData**. Un **multiplexer**, permette di selezionare fra:

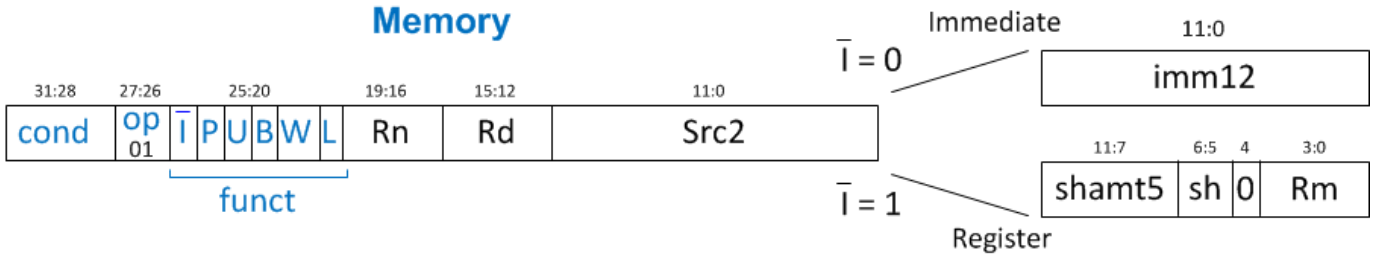
- 0 – **PCPlus4**
- 1 – **ReadData**.

Il segnale di controllo associato al multiplexer è **PCSrc**.



Datapath STR

L'istruzione **STR** scrive una parola di 32 bit contenuta in un registro nella memoria centrale. Il modo in cui questa operazione viene effettuata dipende dalla politica di indirizzamento specificata.

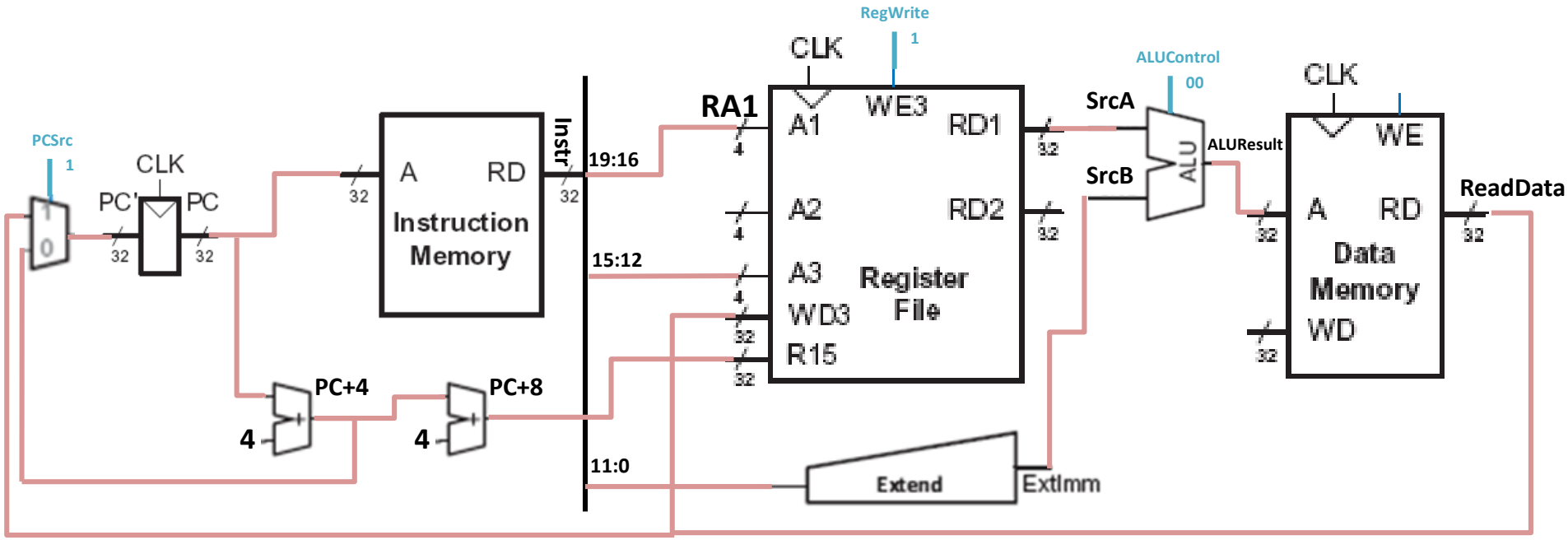


`STR Rd, [Rn, imm12]`

Datapath STR

Estendiamo il **datapath** in modo da poter gestire anche l'istruzioni **STR**.

Come LDR, **STR** legge un indirizzo di base dalla porta **A1** del register file e completa l'immediate. L'**ALU** aggiunge l'indirizzo di base alla costante per trovare l'indirizzo di memoria. Tutte queste funzioni sono già supportate nel datapath.

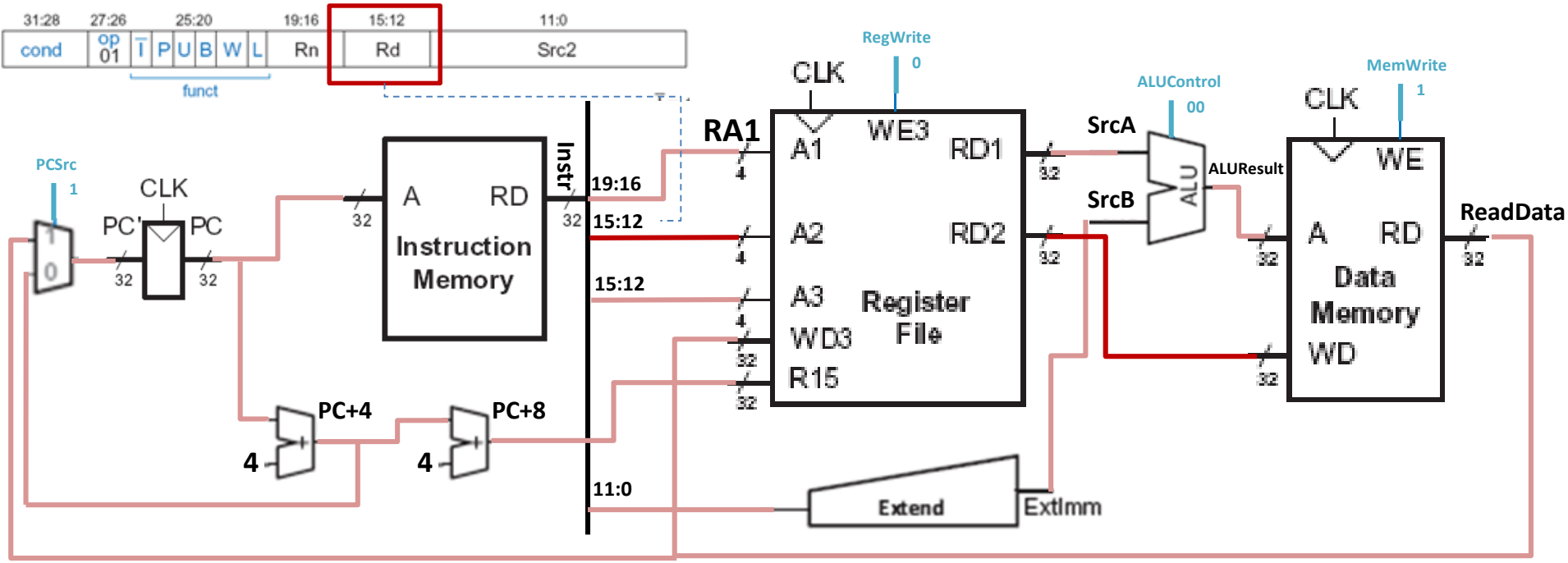


Datapath STR

Il registro è specificato nel campo **Rd**, **Instr$_{15:12}$**, che è collegato alla porta **A2** del register file.

Il valore del registro viene letto sulla porta **RD2**, che è collegata alla porta dati di scrittura (**WD**) della memoria dati.

L'abilitazione del segnale di scrittura **WE** è controllato da **MemWrite**, il quale è 1 se i dati devono essere scritti in memoria.

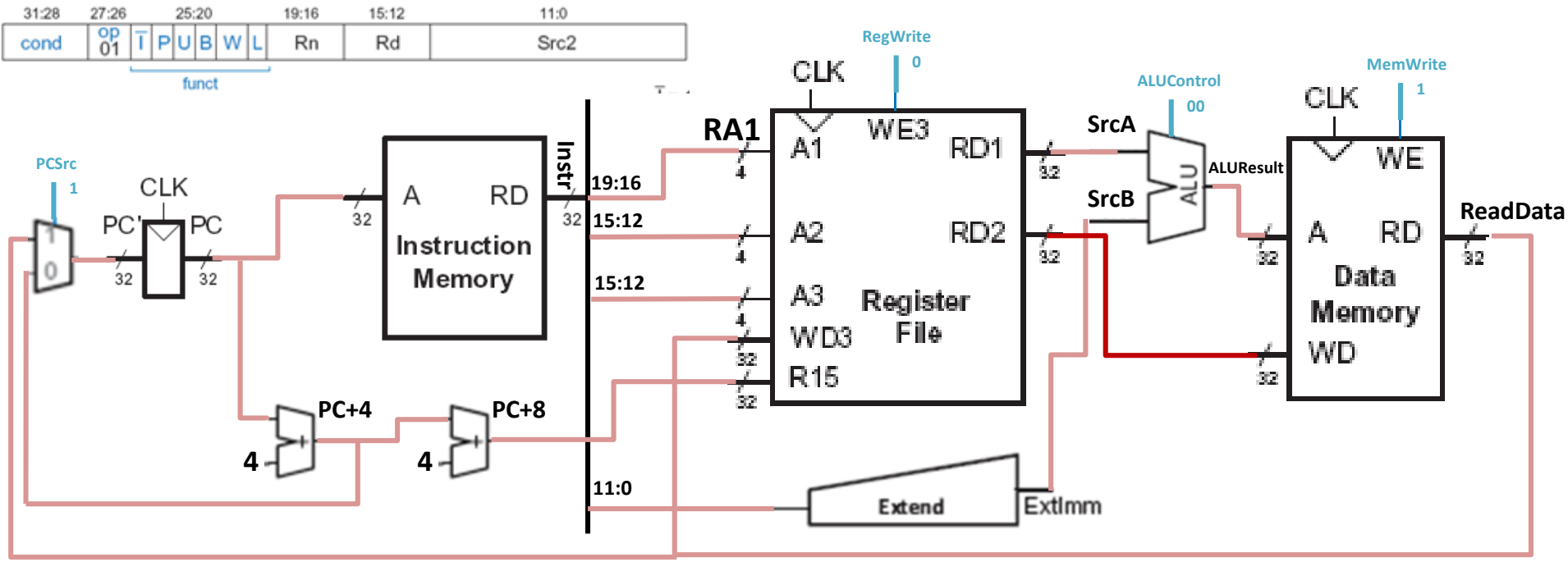


Datapath STR

Il segnale **ALUControl** deve essere impostato a **00** per sommare l'indirizzo di base e l'offset.

Il segnale **RegWrite** è impostato a **0**, perché nulla deve essere scritto nel register file.

Si noti che, pur essendo **Rd** associato anche a A3 e **ReadData** a WD3, questo non produce effetti sul File register, essendo **RegWrite** impostato a **0**.



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 logiche

- `AND`
- `ORR`
- `EOR` **(XOR)**
- `BIC` **(Bit Clear)**
- `MVN` **(MoVe and NOT)**

Source registers

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

Assembly code

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

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 logiche

- Le istruzioni `AND` or `BIC` sono utili per **mascherare** bit
  - **Esempio:** Maschera tutti i bit eccetto il byte meno significativo
  - 0xF234012F `AND` 0x000000FF = 0x0000002F
  - 0xF234012F `BIC` 0xFFFFFF00 = 0x0000002F

- L'istruzione `ORR`: è utile per **combinare** bit fields
  - **Esempio:** combina 0xF2340000 con 0x000012BC:
  - 0xF2340000 `ORR` 0x000012BC = 0xF23412BC

Istruzioni di shifting

- `LSL`: logical shift left
  - **Esempio:** `LSL R0, R7, #5  ; R0=R7 << 5`
- `LSR`: logical shift right
  - **Esempio:** `LSR R3, R2, #31 ; R3=R2 >> 31`
- `ASR`: arithmetic shift right
  - **Esempio:** `ASR R9, R11, #4 ; R9=R11 >>> 4`
- `ROR`: rotate right
  - **Esempio:** `ROR R8, R1, #3  ; R8=R1 ROR 3`

- Lo shift aritmetico a sinistra si comporta come lo shift logico
- Non vi è un'istruzione per la rotazione a sinistra: la rotazione a sinistra di **N** posizioni corrisponde a una rotazione a destra di 32- **N** posizioni
- Lo shift di un valore a sinistra di **N** posizioni è equivalente a moltiplicare tale valore per $\mathbf{2^N}$. Analogamente, uno shift aritmetico aritmetico a destra di un valore **N** è equivalente a dividere per $\mathbf{2^N}$.

Esempi istruzioni di shifting

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

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

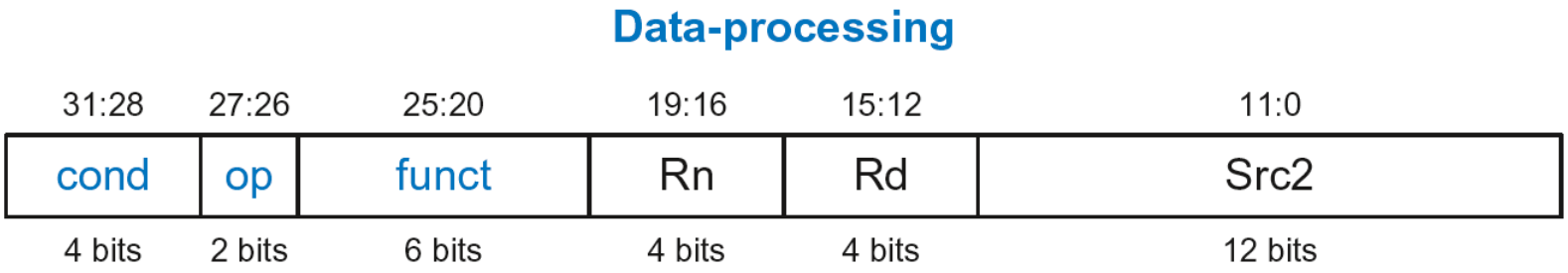
Istruzioni di moltiplicazione

- **`MUL`:** moltiplicazione 32 × 32-bit, risultato 32-bit
  - `MUL R1, R2, R3`
  - **Risultato:** `R1 = (R2 x R3)`$_{31:0}$
- **`UMULL`:** moltiplicazione unsigned 32 × 32-bit, risultato 64-bit
  - `UMULL R1, R2, R3, R4`
  - **Risultato:** `{R1,R4} = R2 x R3` (`R2`, `R3` unsigned)
- **`SMULL`:** moltiplicazione signed 32 × 32-bit, risultato 64-bit
  - `SMULL R1, R2, R3, R4`
  - **Risultato:** `{R1,R4} = R2 x R3` (`R2`, `R3` signed)

Formato istruzioni di data processing

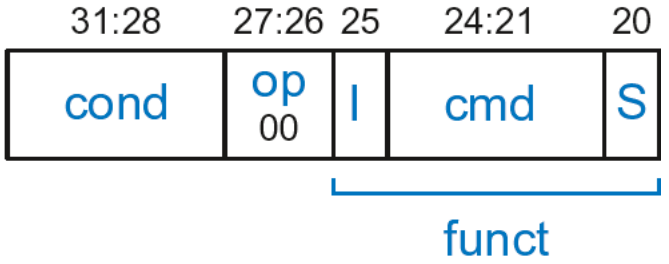
Il formato delle istruzioni di **data-processing** è il più comune.
Il primo operando sorgente è un registro. Il secondo operando sorgente può essere una costante o un registro. La destinazione è un registro.

- **Operandi:**
  - ***Rn*:** primo registro sorgente
  - ***Src2*:** secondo registro sorgente o immediate
  - ***Rd*:** registro destinazione
- **Campi di controllo:**
  - ***cond*:** specifica l'esecuzione condizionale in base ai flag
  - ***op*:** 00 è l'opcode per istruzioni di data processing
  - ***funct*:** specifica il tipo di funzione da eseguire



Formato istruzioni di data processing

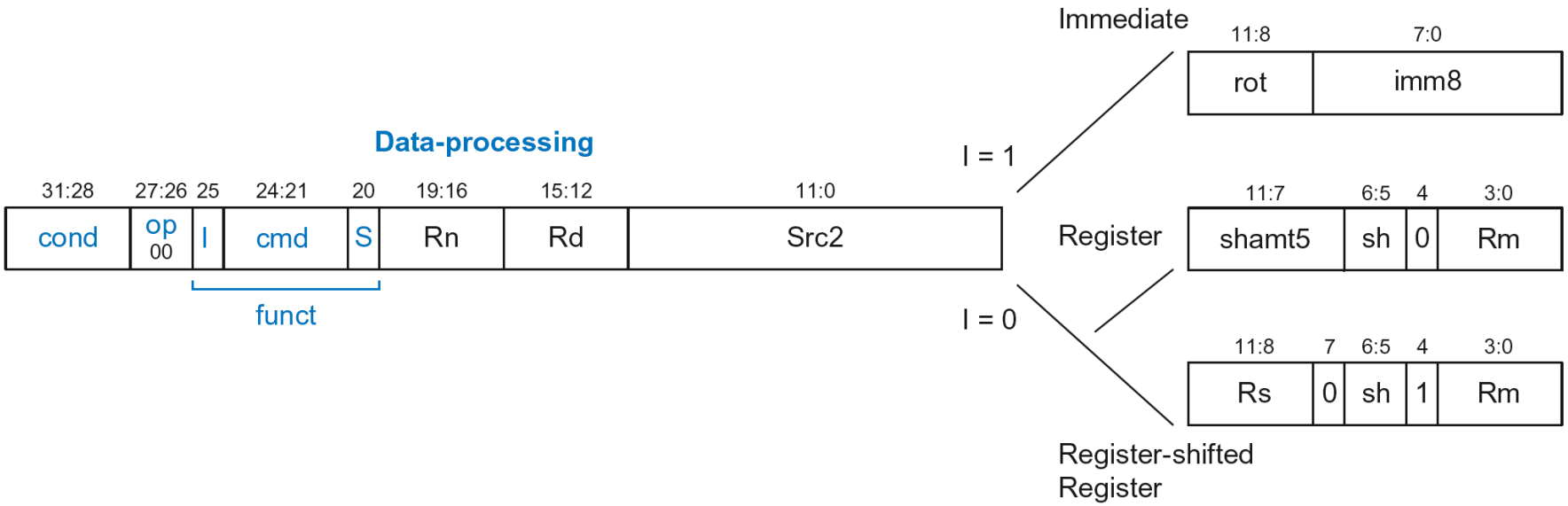
- ***cmd:*** indica l'istruzione specifica da eseguire
  - ***cmd*** = 0100_2 sta per `ADD`
  - ***cmd*** = 0010_2 sta per `SUB`
- ***I***-bit
  - ***I*** = 0: *Src2* è un registro
  - ***I*** = 1: *Src2* è un immediate



- ***S***-bit: quando è 1 allora vengono aggiornate le condition flags
  - ***S*** = 0: `SUB  R0, R5, R7`
  - ***S*** = 1: `ADDS R8, R2, R4` or `CMPS R3, #10`

Formato istruzioni di data processing

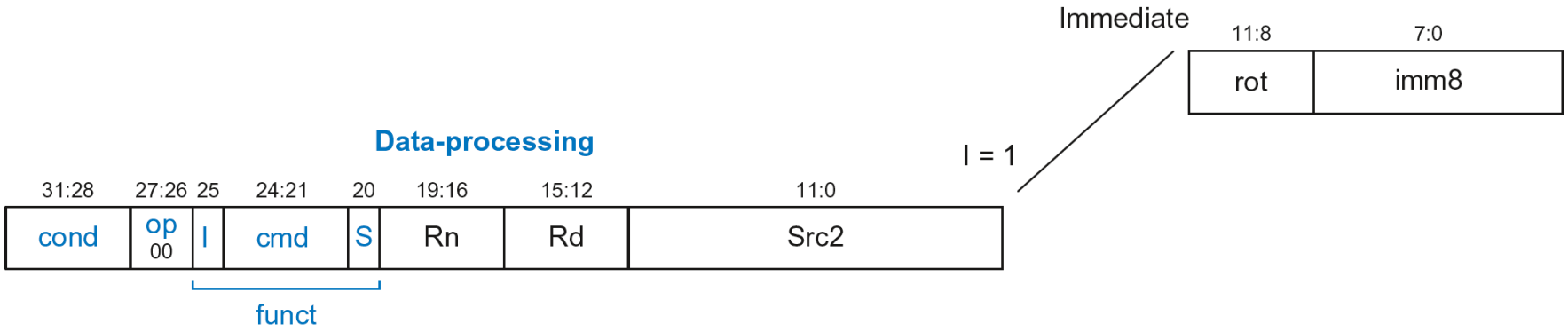
- *Src2* può essere:
  - Immediate
  - Registro
  - Registro “shiftato” da un altro registro



Formato immediate

- **Un immediate è codificato come segue:**
  - *imm8*: 8-bit unsigned immediate
  - *rot*: 4-bit rotation value
- **Da cui si ricava una constante a 32-bit :**

  imm8 **ROR** (*rot* × 2)



Formato immediate

- **Un immediate è codificato come segue:**
  - *imm8*: 8-bit unsigned immediate
  - *rot*: 4-bit rotation value
- **Da cui si ricava una constant a 32-bit :**

$$imm8 \textbf{ ROR } (rot \times 2)$$

Esempio: *imm8* = 10001111

rot	32-bit constant
0000	0000 0000 0000 0000 0000 0000 1000 1111
0001	1100 0000 0000 0000 0000 0000 0010 0011
0010	1111 0000 0000 0000 0000 0000 0000 1000
…	…
1111	0000 0000 0000 0000 0000 0010 0011 1100

Esempio: Add

```
ADD R0, R1, #42
```

cond = 1110_2 (14) indica un'esecuzione non condizionata
op = 00_2 (0) indica un'istruzione di data-processing
cmd = 0100_2 (4) è il codice di ADD
I = 1 indica che ***Src2*** è un immediate, ***S***= 0,
Rd = 0, ***Rn*** = 1
imm8 = 42, ***rot*** = 0

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
cond	op	I	cmd	S	Rn	Rd	shamt5	sh Rm
1110	00	1	0100	0	0001	0000	0000	00101010

0xE281002A

Esempio: SUB

```
SUB R2, R3, #0xFF0
```

cond = 1110_2 (14)
op = 00_2 (0)
cmd = 0010_2 (2) è il codice di SUB
I=1, ***S***=0
Rd = 2, ***Rn*** = 3
imm8 = 0xFF
per produrre 0xFF0 ***imm8*** deve essere ruotato di 4 bit a sinistra, ovvero 28 bit a destra. Quindi, ***rot*** = 14

Field Values

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7:0
1110_2	00_2	1	0010_2	0	3	2	14	255
cond	op	I	cmd	S	Rn	Rd	rot	imm8
1110	00	1	0010	0	0011	0010	1110	11111111

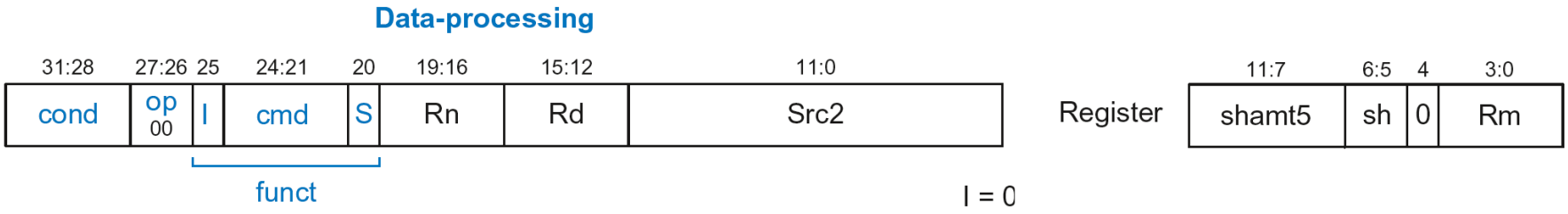
0xE2432EFF

Formato register

***Rm*:** indica il registro che fa da secondo operado

***shamt5*:** indica di quanto il valore in Rm è shiftato

***sh*:** indica il tipo di shift (i.e., >>, <<, >>>, ROR)



Shift Type	*sh*
LSL	00_2
LSR	01_2
ASR	10_2
ROR	11_2

Esempio: Add

```
ADD R5, R6, R7
```

Operation: R5 = R6 + R7
cond = 1110_2 (14)
op = 00_2 (0)
cmd = 0100_2 (4)
I=0 indica che ***Src2*** è un registro
Rd = 5, ***Rn*** = 6, ***Rm*** = 7
shamt = 0, ***sh*** = 0

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
cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm
1110	00	0	0100	0	0110	0101	00000	00	0	0111

0xE0865007

Esempio: OR

`ORR R9, R5, R3,` **`LSR #2`**

Operation: R9 = R5 OR (R3 >> 2)

cond = 1110_2 (14)

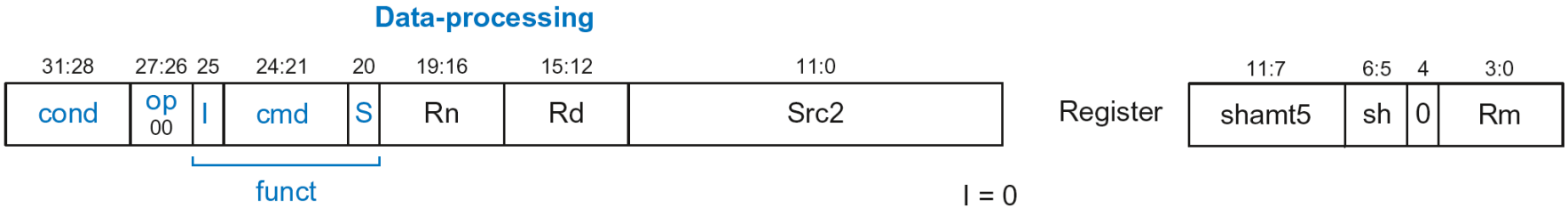
op = 00_2 (0)

cmd = 1100_2 (12) indica l'istruzione `ORR`

I=0

Rd = 9, ***Rn*** = 5, ***Rm*** = 3

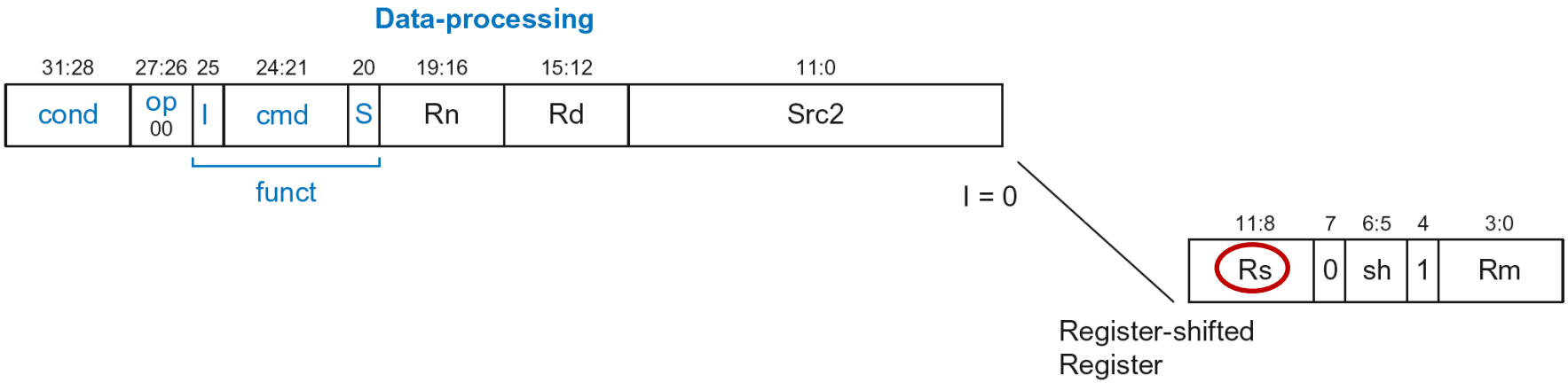
shamt5 = 2, ***sh*** = 01_2 (LSR)



1110 00 0 1100 0 0101 1001 00010 01 0 0011

0xE1859123

Formato register-shifted register



Simile al formato register, con la differenza che il numero di posizioni di cui Rm deve shiftare non è una costante ma è indicato dal registro Rs.

Esempio: XOR

```
EOR R8, R9, R10, ROR R12
```

Operation: R8 = R9 XOR (R10 ROR R12)

cond = 1110_2 (14)

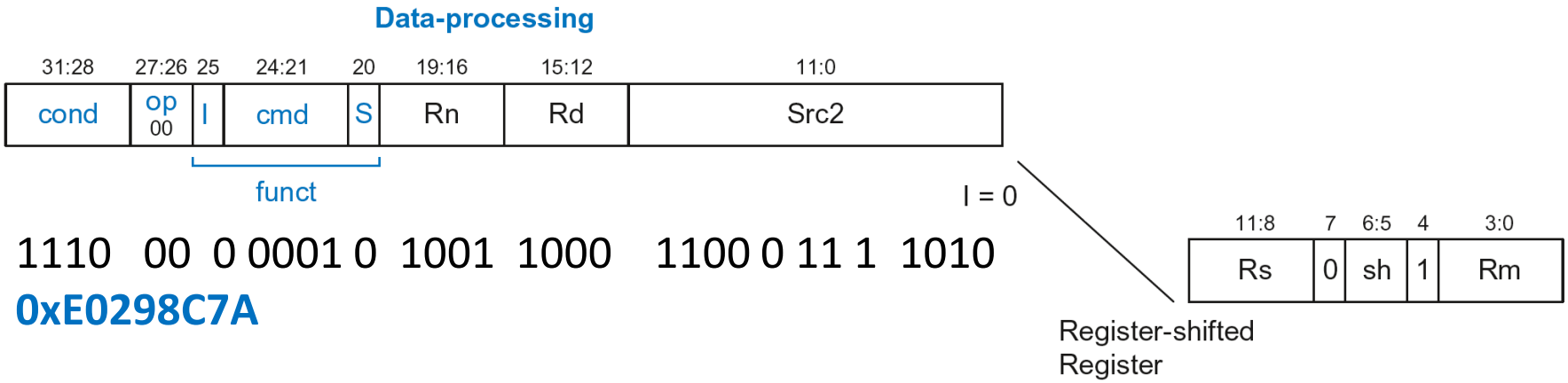
op = 00_2 (0)

cmd = 0001_2 (1) indica l'instruzione `EOR`

I=0

Rd = 8, ***Rn*** = 9, ***Rm*** = 10, ***Rs*** = 12

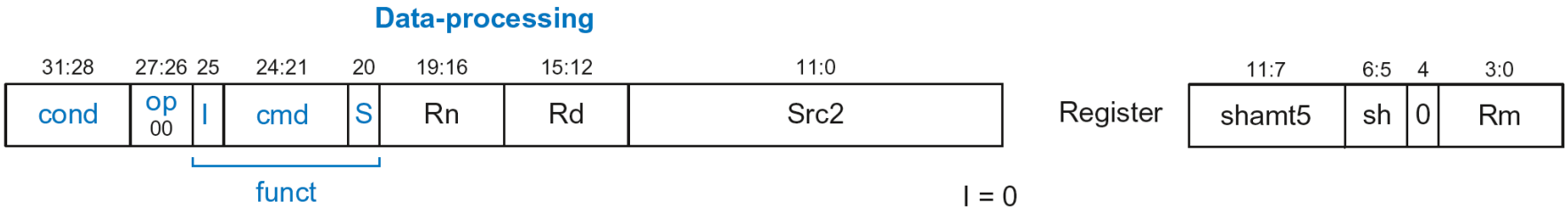
sh = 11_2 (ROR)



Esempio: ROR

```
ROR R1, R2, #23
```

Operation: R1 = R2 ROR 23
cond = 1110_2 (14)
op = 00_2 (0)
cmd = 1101_2 (13) codice usato per tutti I tipi di shift (LSL, LSR, ASR, ROR)
Src2 è un immediate-shifted register quindi ***I***=0
Rd = 1, ***Rn*** = 0, ***Rm*** = 2
shamt5 = 23, ***sh*** = 11_2 (ROR)



1110 00 0 1101 0 0000 0001 10111 11 0 0010

0xE1A01BE2

Datapath ADD, SUB, AND, ORR

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**.

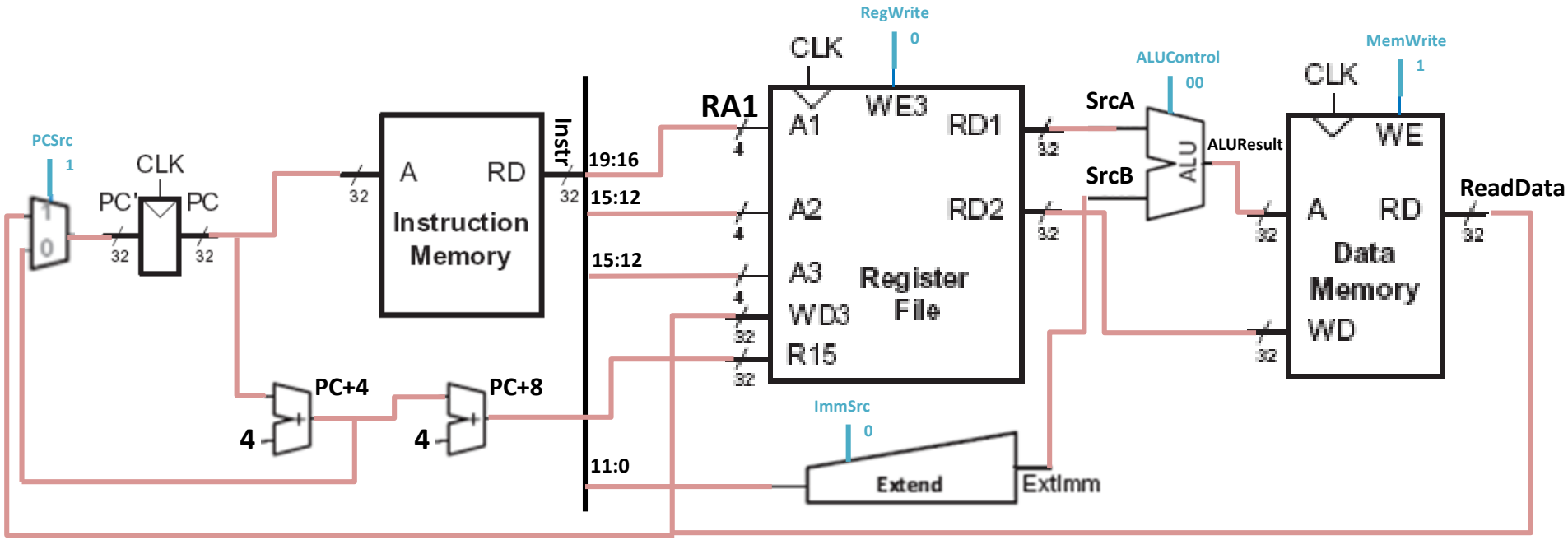
L'ALU imposta anche dei bit in **ALUFlags**$_{3:0}$:

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

Datapath ADD, SUB, AND, ORR

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**);



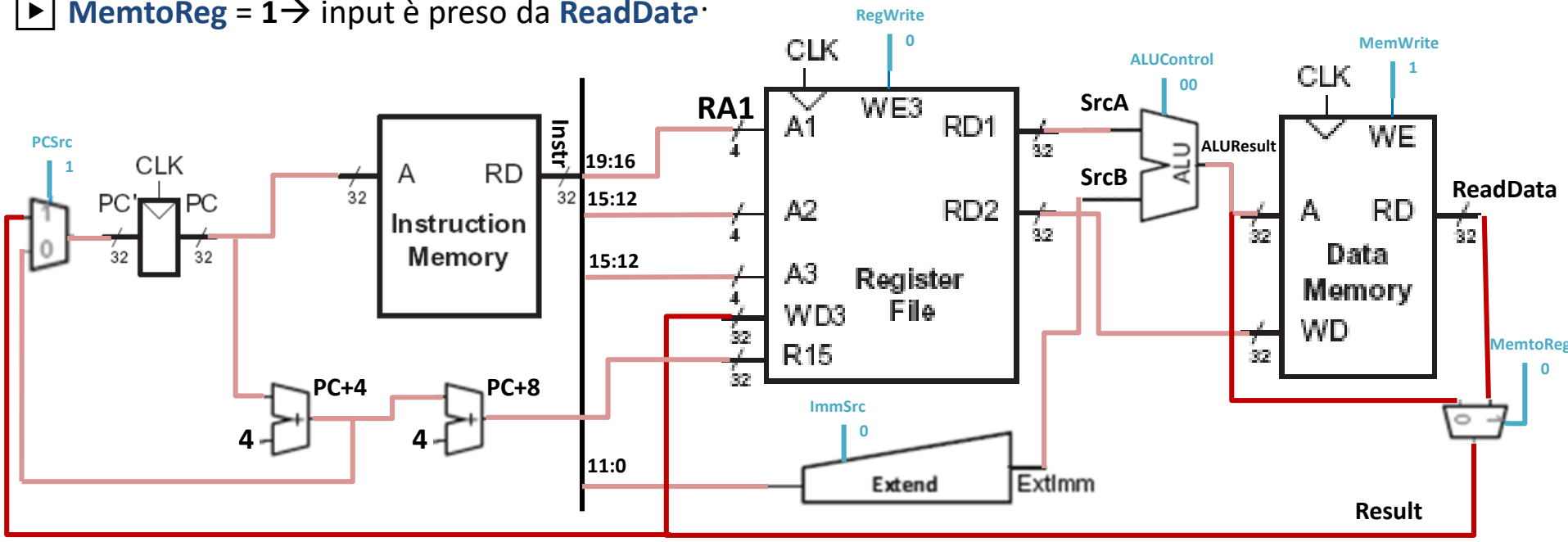
Datapath ADD, SUB, AND, ORR

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**.

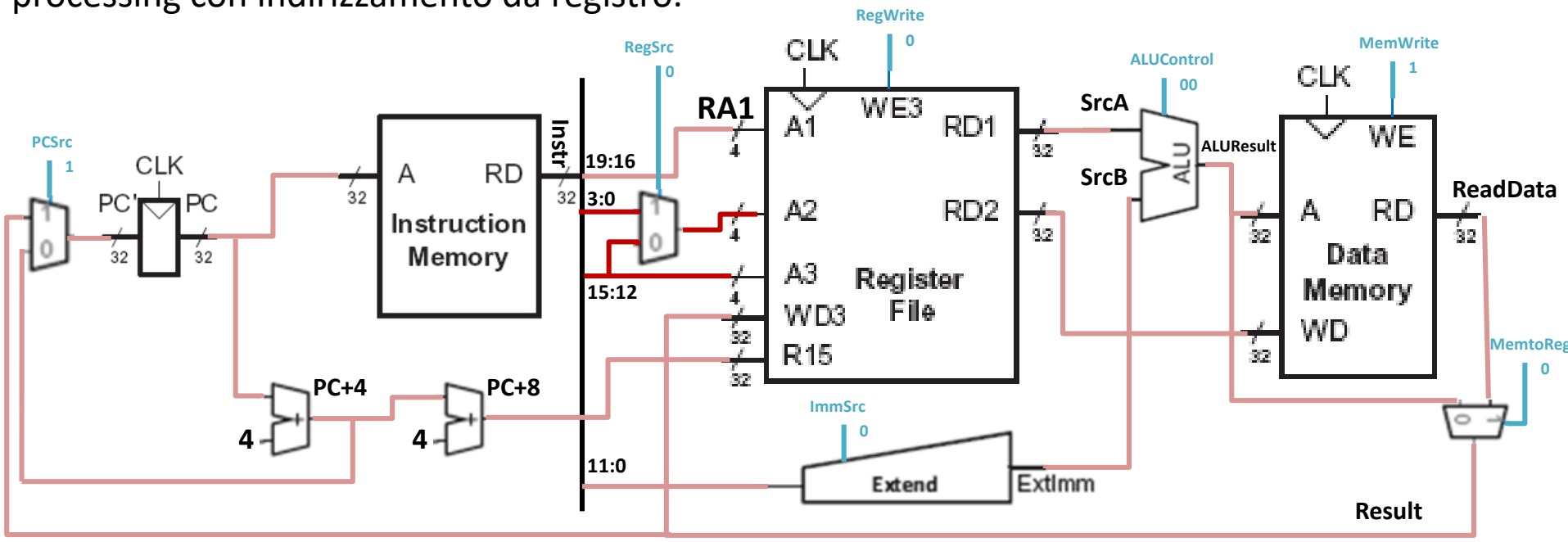


Datapath ADD, SUB, AND, ORR

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:

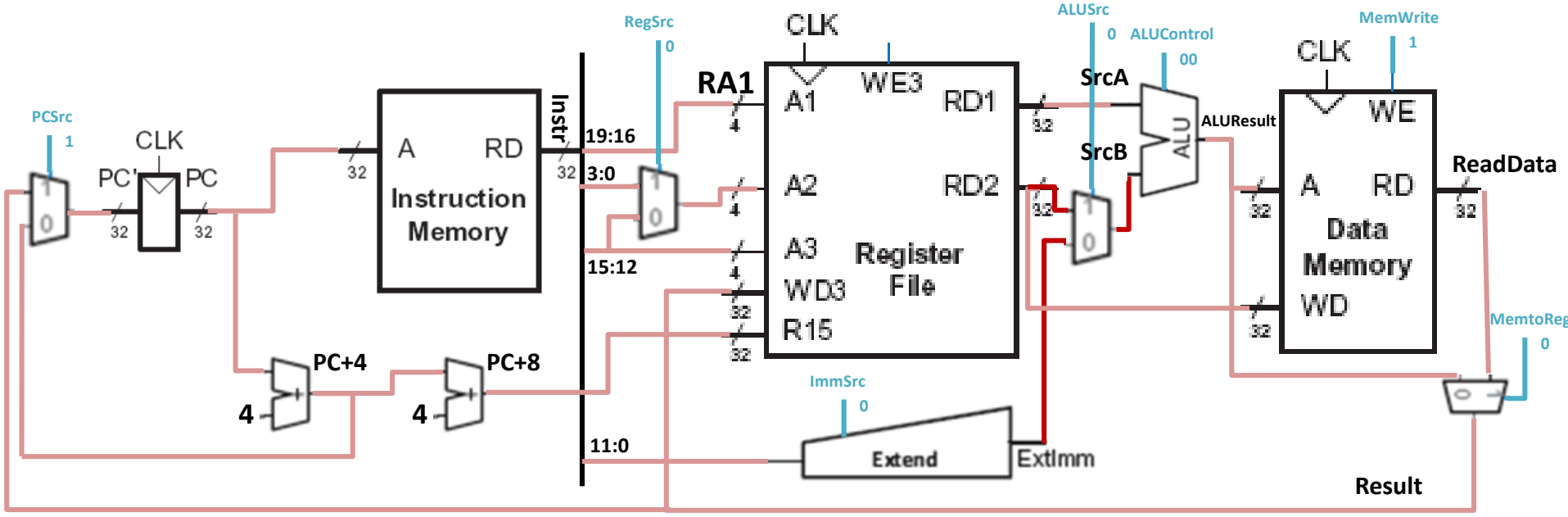
- **Rd** (**$Instr_{15:12}$**) per **STR**;
- **Rm** (**$Instr_{3:0}$**) per istruzioni di data processing con indirizzamento da registro.



Datapath ADD, SUB, AND, ORR

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.



Esecuzione condizionata

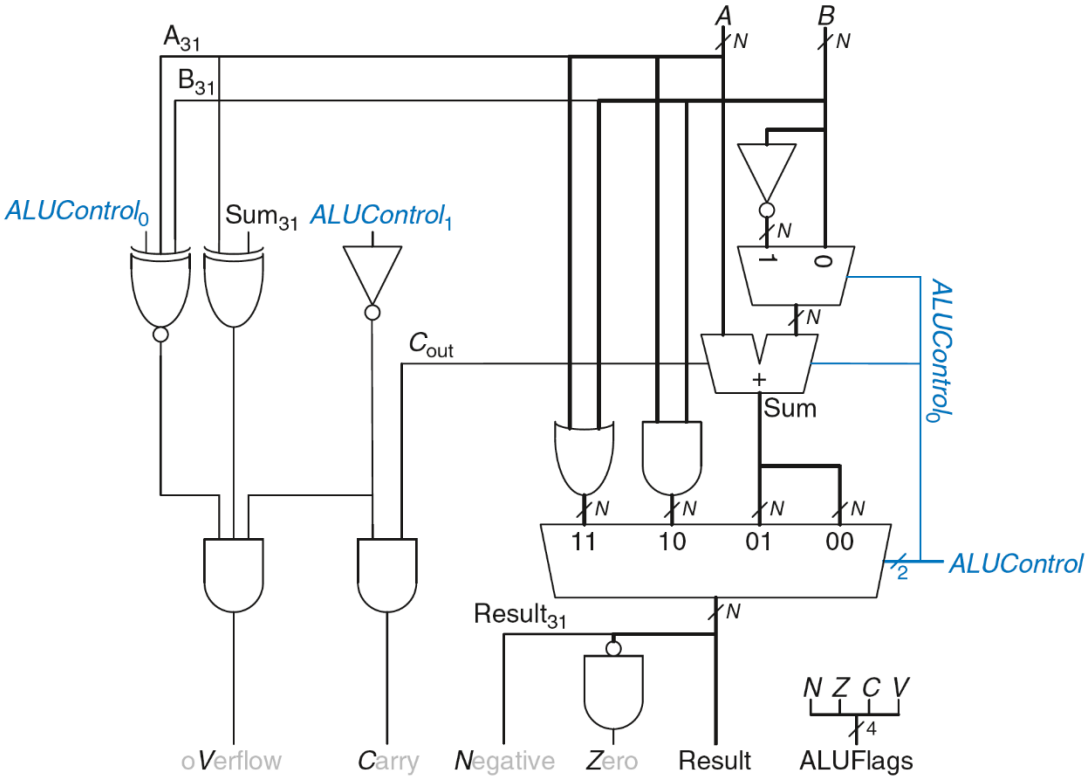
A volte si vuole che l'esecuzione di una istruzione dipenda dal verificarsi o meno di una certa condizione, per esempio:

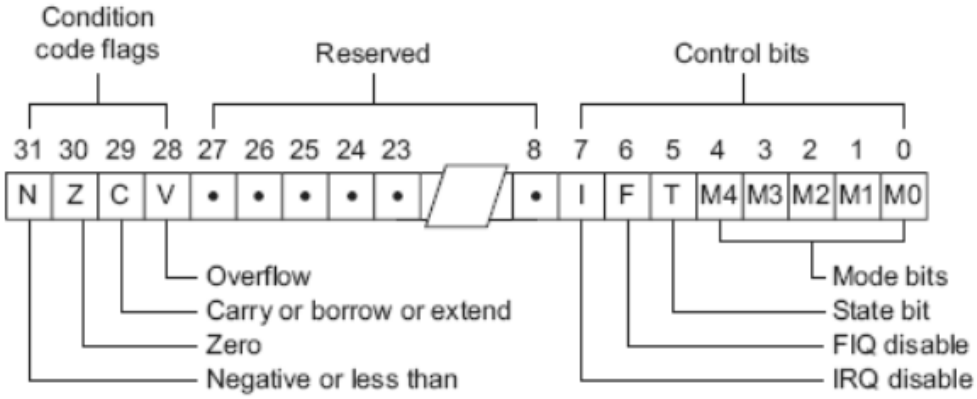
- Istruzioni if/then/else, while, etc.: un blocco di istruzioni è eseguito solo se una data condizione valuta a true

ARM usa delle **condition flag** che sono

- memorizzate nel *Current Program Status Register* (*CPSR*)
- settate da una istruzione
- usate per eseguire in modo condizionato una istruzione

Flag	Name	Description
N	**N**egative	Instruction result is negative
Z	**Z**ero	Instruction results in zero
C	**C**arry	Instruction causes an unsigned carry out
V	o**V**erflow	Instruction causes an overflow


A
B
N
A31
B31
ALUControl0
Sum31
ALUControl1
Cout
1
0
+
Sum
ALUControl0
11
10
01
00
2
ALUControl
Result31
N Z C V
4
oVerflow
Carry
Negative
Zero
Result
ALUFlags


Condition code flags
Reserved
Control bits
31 30 29 28 27 26 25 24 23 8 7 6 5 4 3 2 1 0
N Z C V • • • • • • I F T M4 M3 M2 M1 M0
Overflow
Carry or borrow or extend
Zero
Negative or less than
Mode bits
State bit
FIQ disable
IRQ disable

Istruzione di comparazione: CMP

```
CMP R5, R6
```

- Esegue: R5-R6
- Non salva il risultato
- Aggiorna i flag nel Current Program Status Register. Se il risultato di R5-R6:
  - è uguale a 0, allora *Z*=1
  - è negativo, allora *N*=1
  - causa un carry out, *C*=1
  - causa un signed overflow, *V*=1

Suffisso S al nome di una istruzione

```
ADDS R1, R2, R3
```

- Esegue: R2 + R3
- Salva il risultato in R1
- Aggiorna i flag nel Current Program Status Register. Se il risultato di R2 + R3:
  - è uguale a 0, allora *Z*=1
  - è negativo, allora *N*=1
  - causa un carry out, *C*=1
  - causa un signed overflow, *V*=1

Istruzioni condizionate

- Un istruzione può essere ***eseguita condizionalmente*** sulla base dei valori delle condition flags
- In assembly, un'esecuzione condizionata è indicata da un suffisso detto ***condition mnemonic.***

```
CMP    R1, R2
SUBNE  R3, R5, R8
```

- **NE:** `SUB` verrà eseguito solo se R1 ≠ R2, ovvero Z = 0

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 $NZCV = 0011_2$

+ 1110 HS: TRUE

10111 GE: FALSE

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

(b)

A – B: 0101 $NZCV = 1001_2$

+ 0011 HS: FALSE

1000 GE: TRUE

Esempi di istruzioni condizionate

```
CMP   R5, R9          ; performs R5-R9
                      ; sets condition flags

SUBEQ R1, R2, R3      ; executes if R5==R9 (Z=1)
ORRMI R4, R0, R9      ; executes if R5-R9 is
                      ; negative (N=1)
```

Si assuma per esempio che R5 = 17, R9 = 23:

- `CMP` esegue: 17 – 23 = -6 (flags: *N*=1, *Z*=0, *C*=0, *V*=0)
- `SUBEQ` **non è eseguita** (non sono uguali: *Z*=0)
- `ORRMI` **è eseguita** poiché il risultato è negativo (*N*=1)

Istruzioni di Branching

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 branching 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.

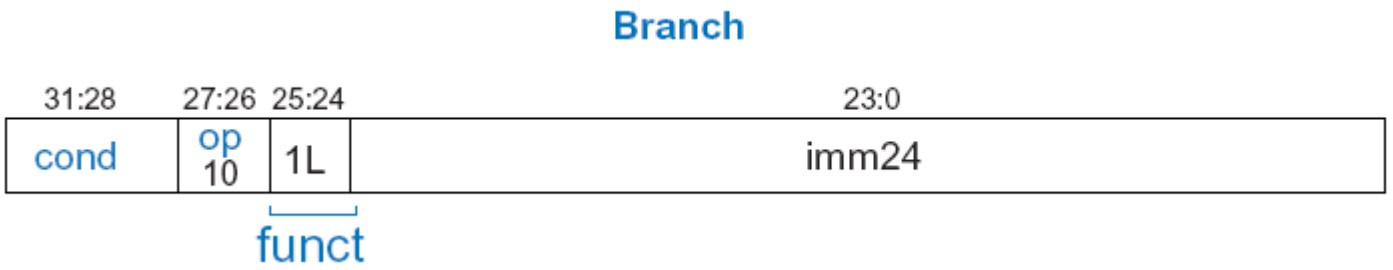
Istruzioni di Branching

Le istruzioni di branching 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.



Istruzioni di Branching

La costante a **24**-bit *viene moltiplicata per 4* ed estesa con segno. Pertanto, la logica **Extend** necessita di una ulteriore modalità. **ImmSrc** è, quindi, esteso a 2 bit.

L'istruzione di salto somma poi **ImmSrc** a **PC+8** e scrive il risultato di nuovo nel **PC**.

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 Branching

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

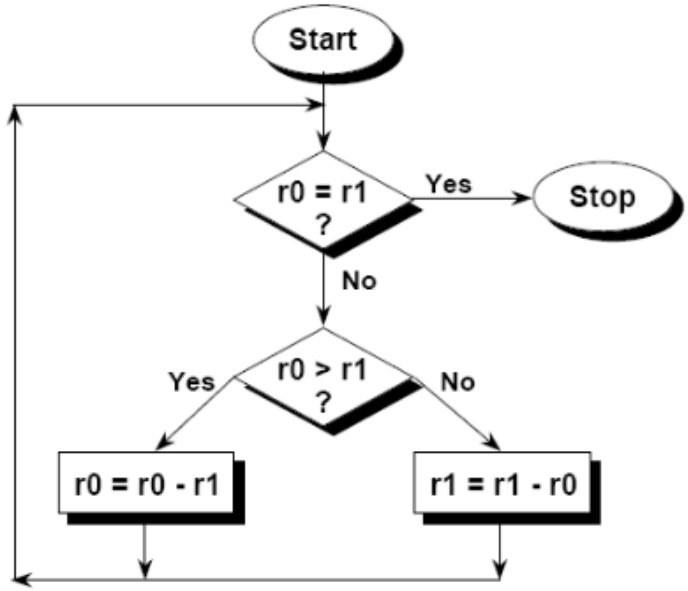
Il registro **R14** ha la funzione (architetturale) di subroutine Link Register (**LR**).

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
```

Esempi di istruzioni di branching

Il processore ARM consente un codice molto compatto.



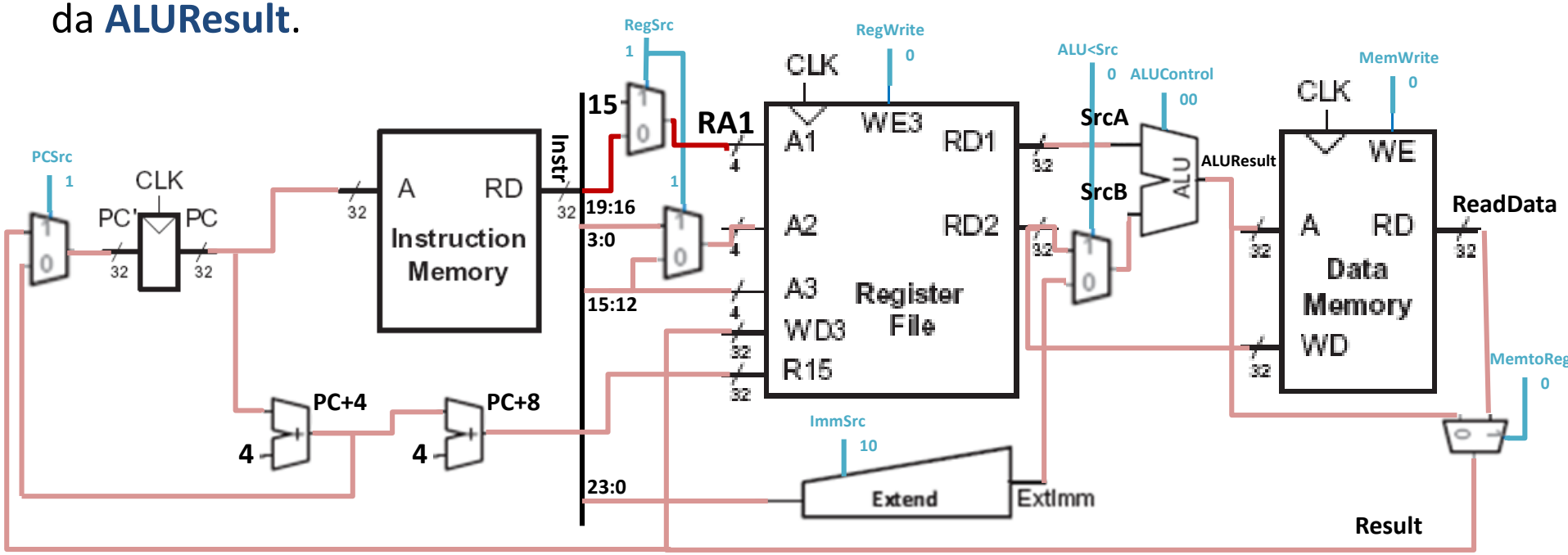

```
MCD   cmp r0,r1          ;if r0 > r1
      subgt r0,r0,r1     ;then r0 <- r0-r1
      sublt r1,r1,r0     ;else r1 <- r1-r0
      bne MCD            ;raggiunta la fine?
```

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

Datapath istruzioni di branching

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**.

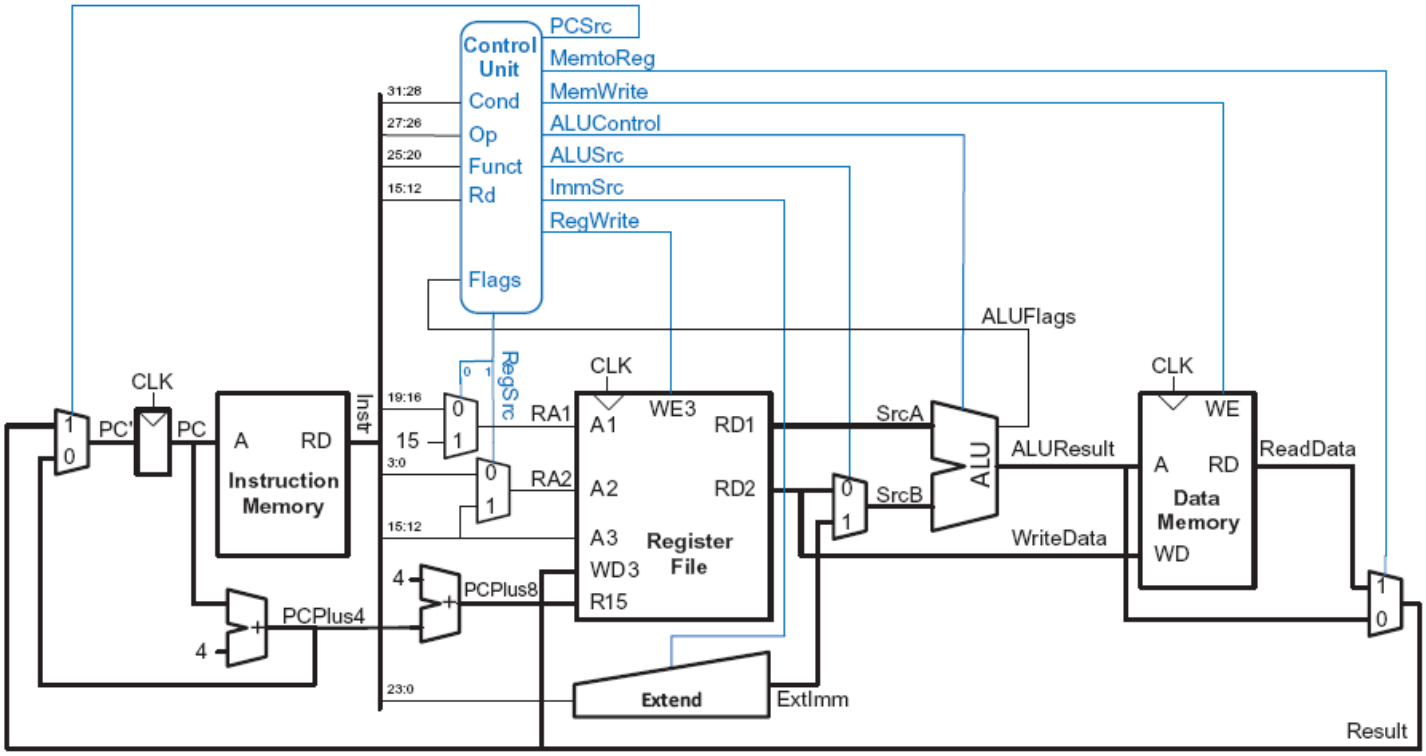


Unità di controllo

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 nel *Current Program Status Register* e li aggiorna in modo appropriato.



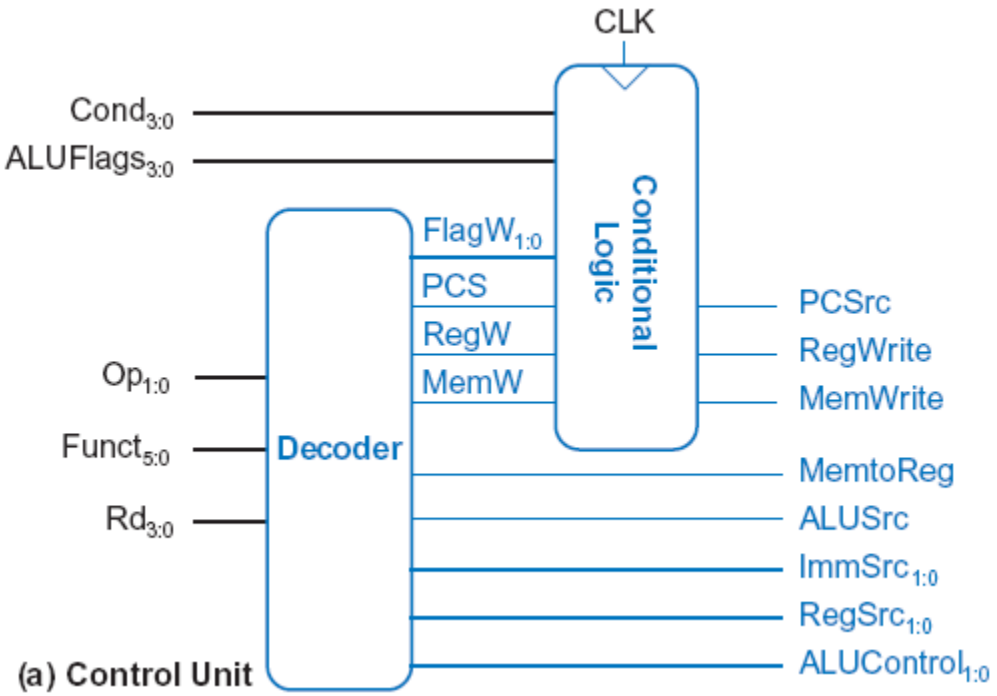
Unità di controllo

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:

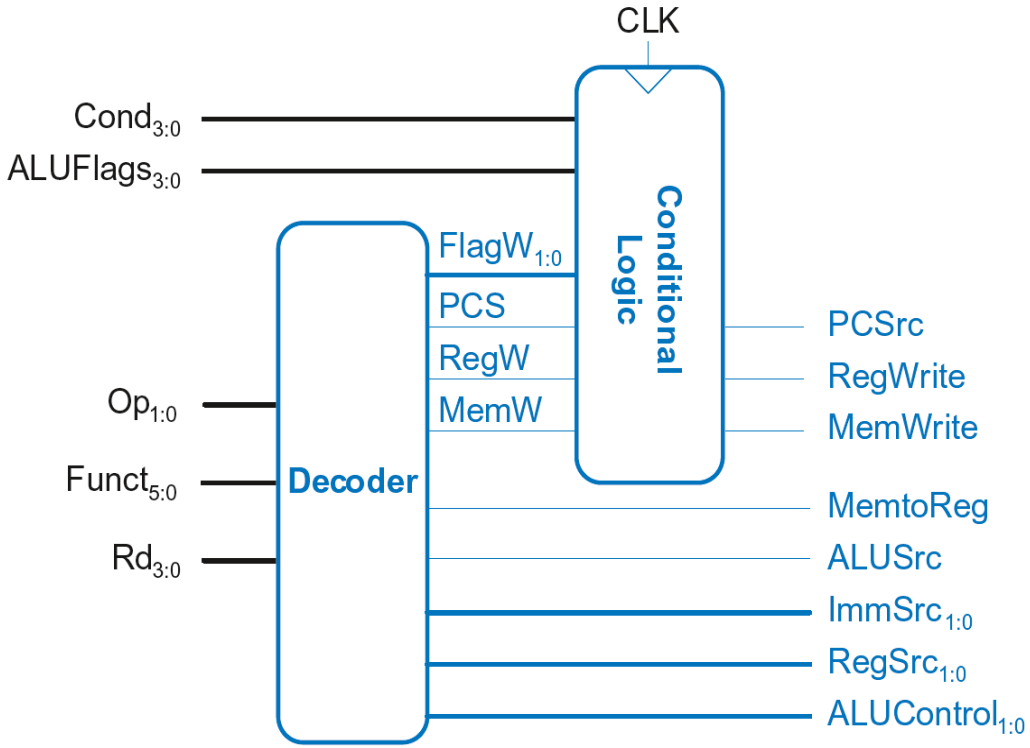
- un **decodificatore principale**, che produce la maggior parte dei segnali di controllo;
- un **decoder ALU**, che utilizza il campo Funct per determinare il tipo di istruzione data-processing;
- la **logica di controllo del PC**, che determina se il PC deve essere aggiornato a causa di una istruzione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



Unità di controllo

Dividiamo l'unità di controllo in due parti principali:

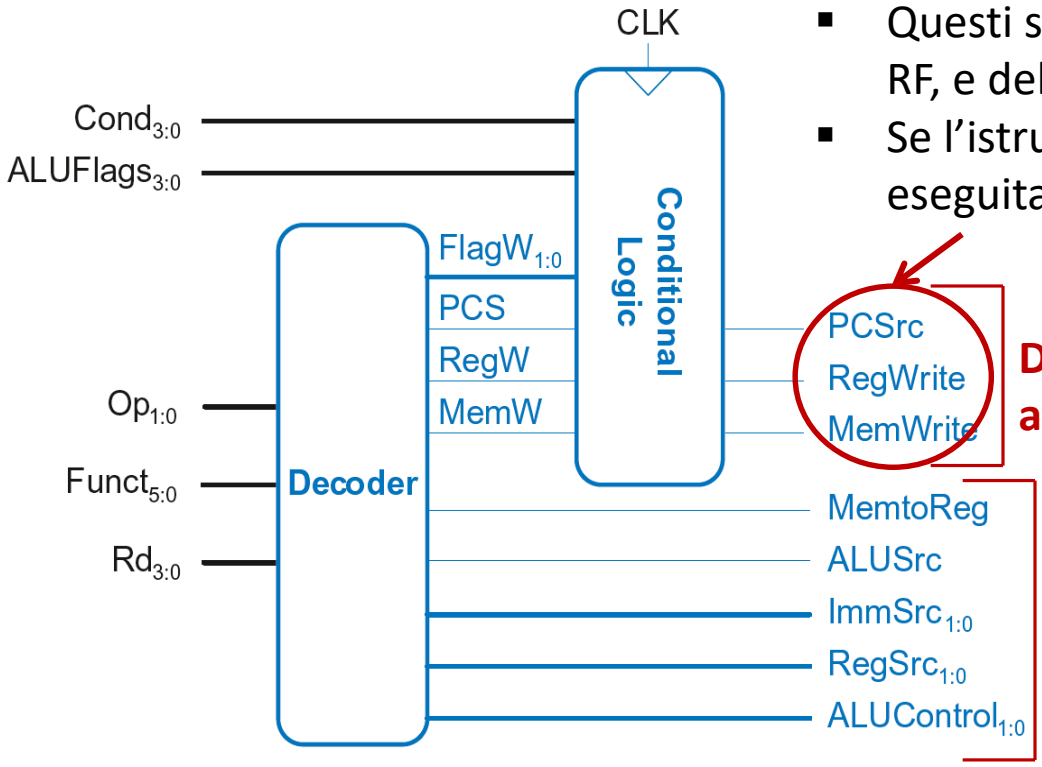
- il **decoder** – genera i segnali di controllo sulla base dei campi di Instr;
- la **logica condizionale** – gestisce i flag di stato e li aggiorna quando l'istruzione deve essere eseguita su condizione.



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.

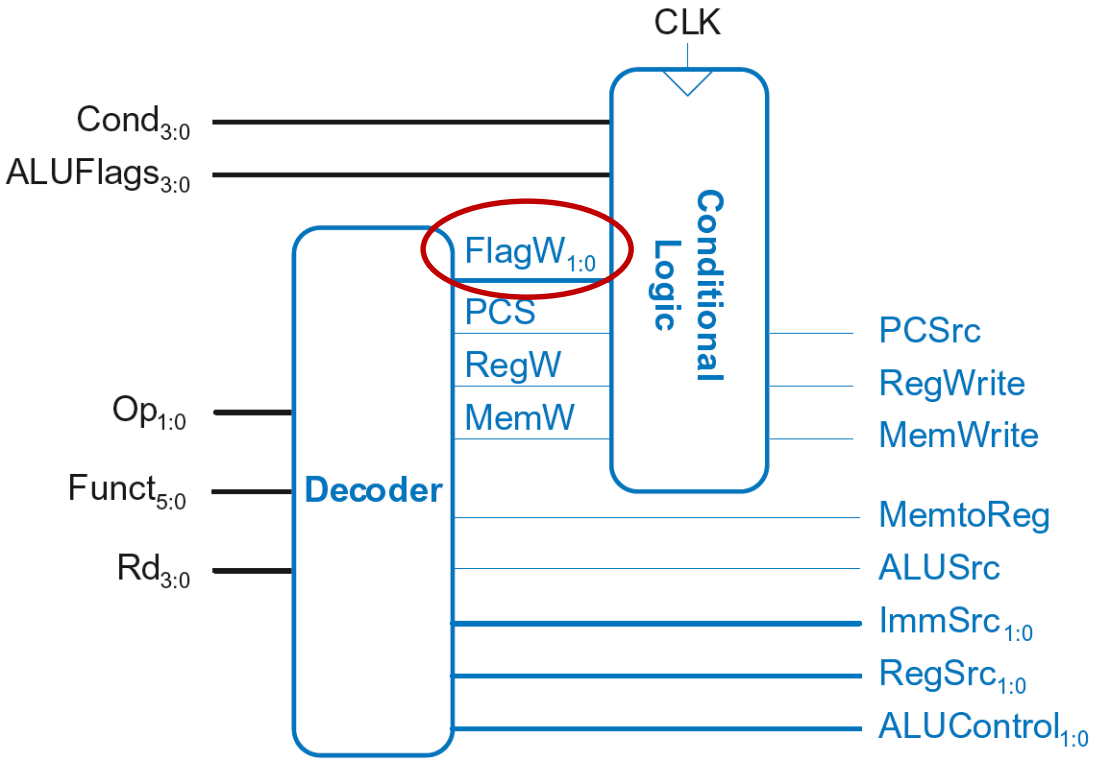


- Questi segnali cambiano lo stato di PC, RF, e della memoria
- Se l'istruzione non deve essere eseguita sono forzati a 0

Diretti al datapath attraverso la logica condizionale

Direttamente al datapath

Unità di controllo



- ***FlagW*$_{1:0}$:** Flag Write signal, indica quando le *ALUFlags* devono essere aggiornate, ovvero quando in una istruzione S=1
- `ADD`, `SUB` aggiornano tutti i flag (**NZCV**)
- `AND`, `ORR` aggiornano solo **N** e **Z**
- Quindi sono necessari due bit:
  - ***FlagW*$_1$** = 1: *NZ* (*ALUFlags*$_{3:2}$ saved)
  - ***FlagW*$_0$** = 1: *CV* (*ALUFlags*$_{1:0}$ saved)

Decoder

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

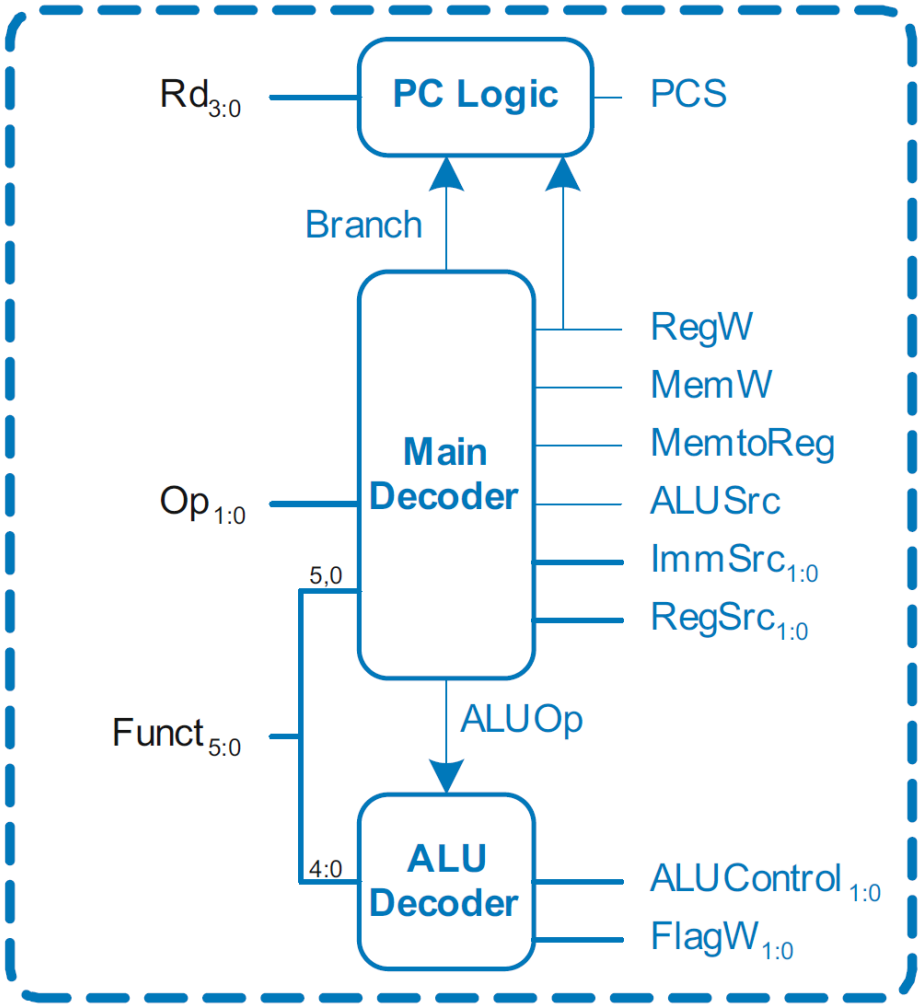
▶ produrre i segnali di controllo adeguati per il datapath. Alcuni segnali sono inviati direttamente al datapath: **MemtoReg**, **ALUSrc**, **ImmSrc1:0**, e **RegSrc1:0**.

▶ 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.

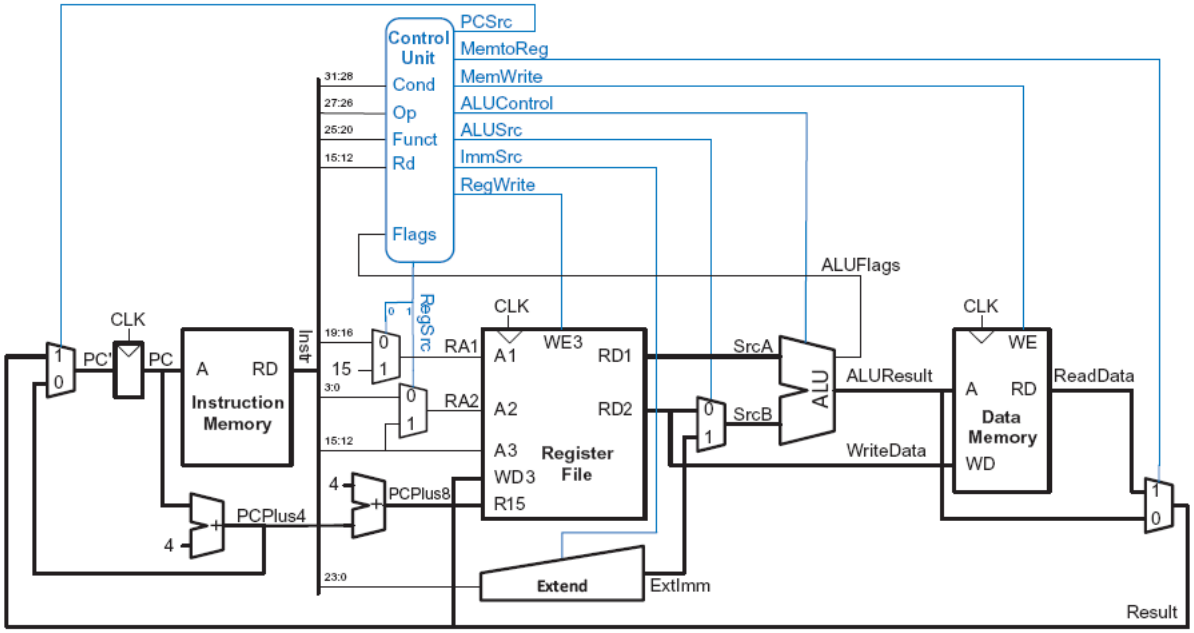
▶ generare i segnali **Branch** e **ALUOp**, utilizzati rispettivamente per indicare l'istruzione B o il tipo di istruzione data processing.

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

- **Main Decoder**
- ALU Decoder
- PC Logic

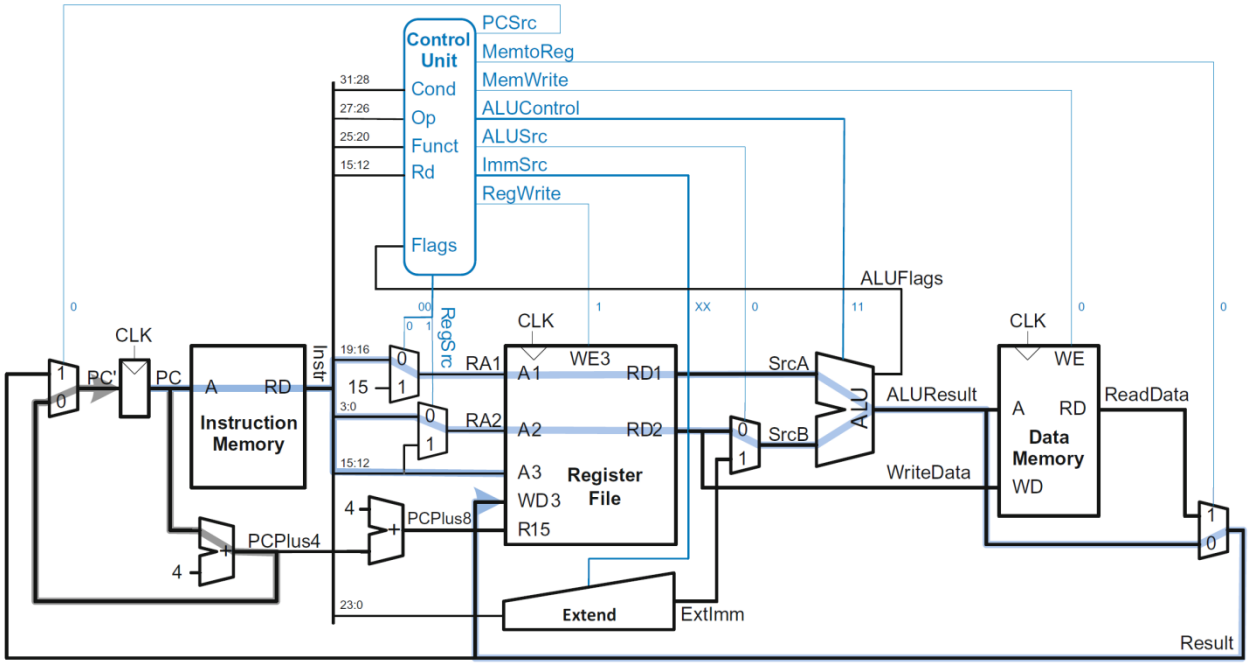


Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
11	X	X	B	1	0	0	1	10	0	X1	0

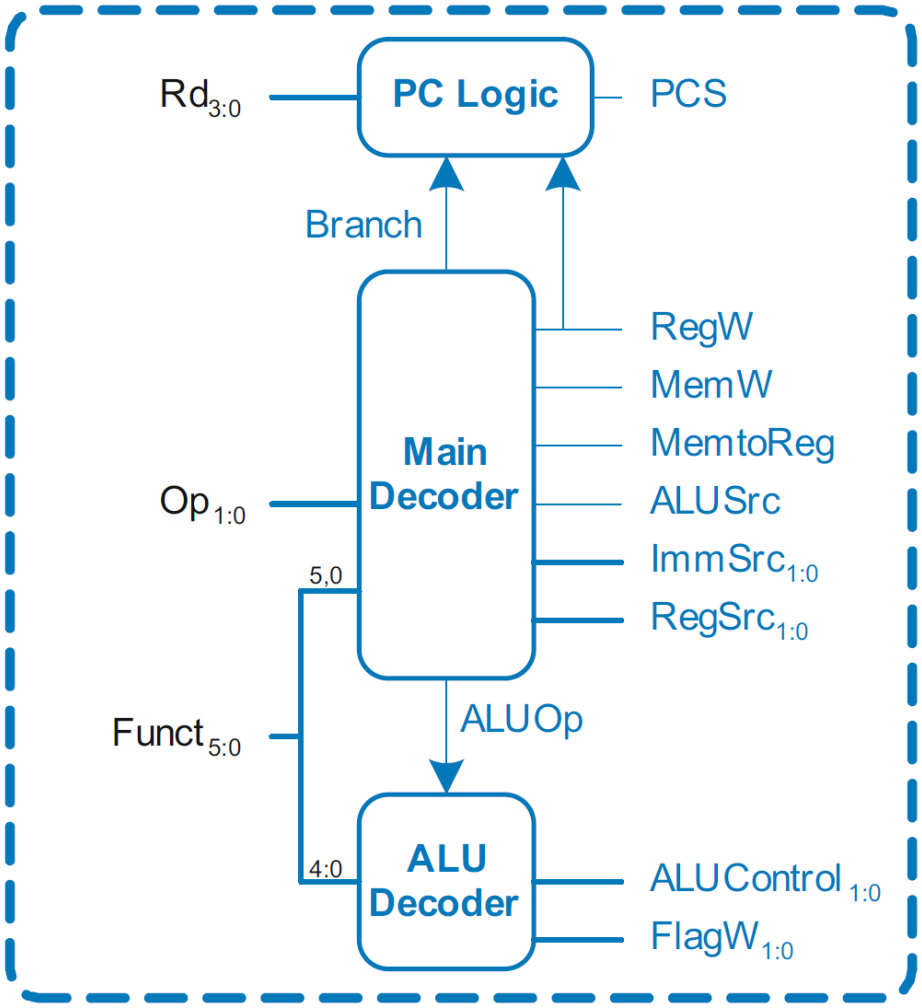


Esempio: DPReg

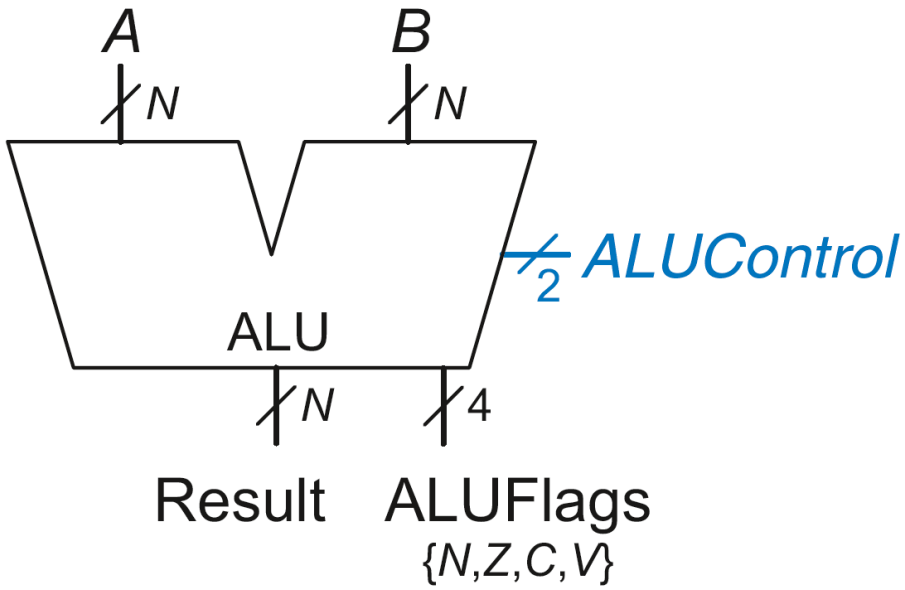
Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1



- Main Decoder
- **ALU Decoder**
- PC Logic



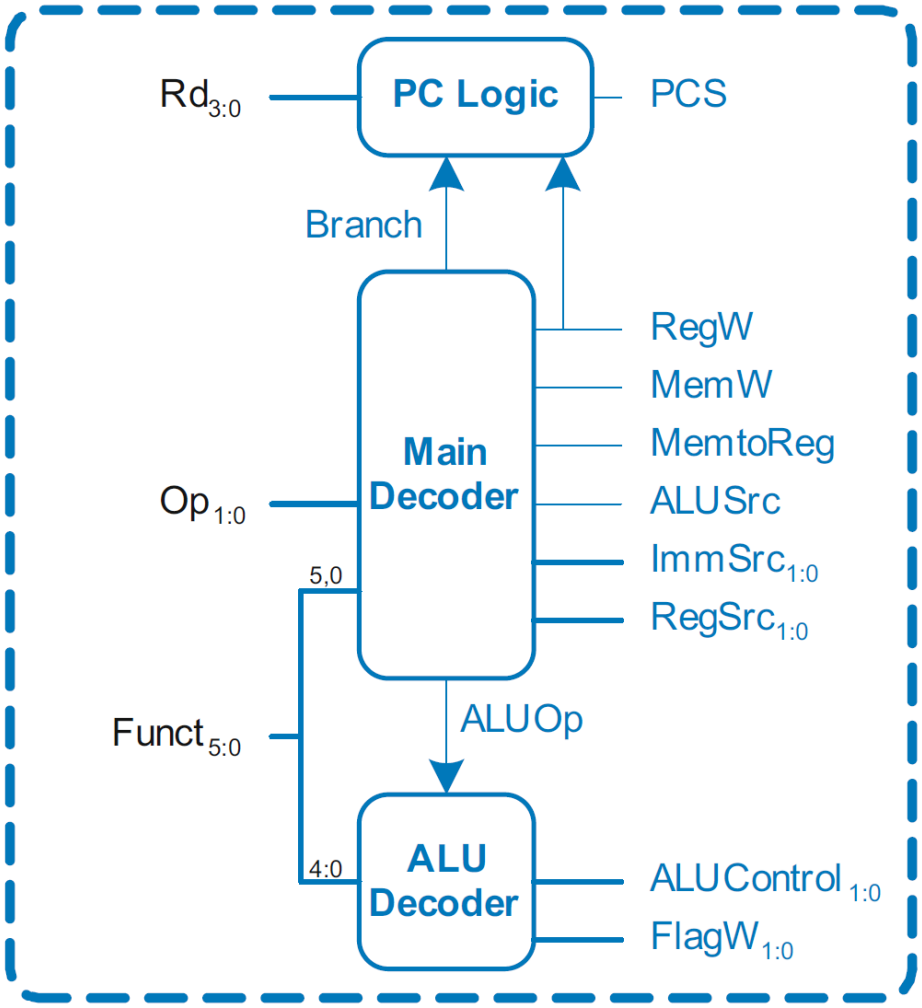
$ALUControl_{1:0}$	Function
00	Add
01	Subtract
10	AND
11	OR



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

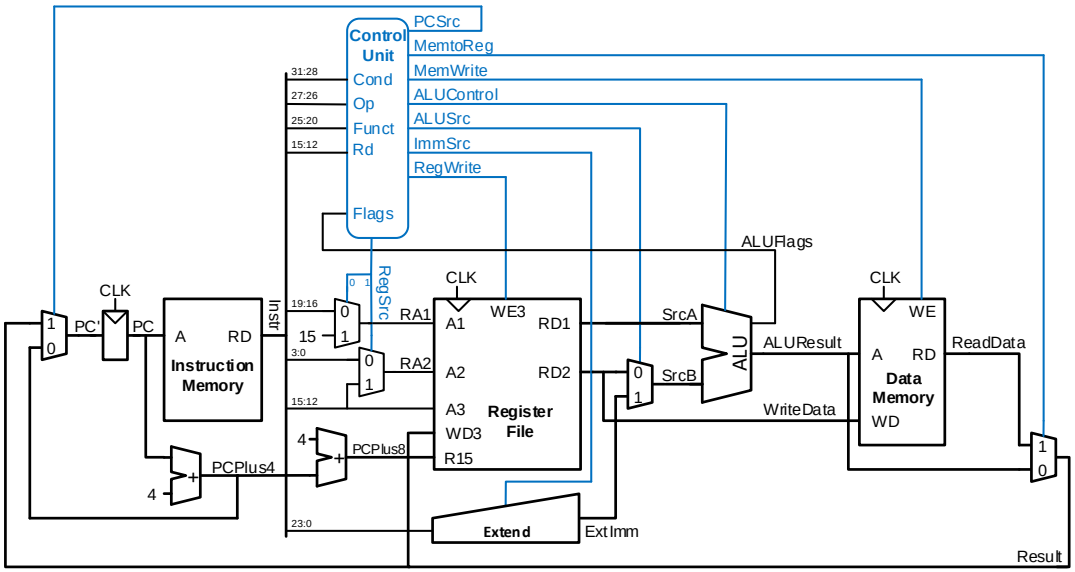
- $FlagW_1$ = 1: *NZ* ($Flags_{3:2}$) devono essere salvate
- $FlagW_0$ = 1: *CV* ($Flags_{1:0}$) devono essere salvate

- Main Decoder
- ALU Decoder
- **PC Logic**



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

$$PCS = ((Rd == 15)\ \text{AND}\ RegW)\ \text{OR}\ Branch$$

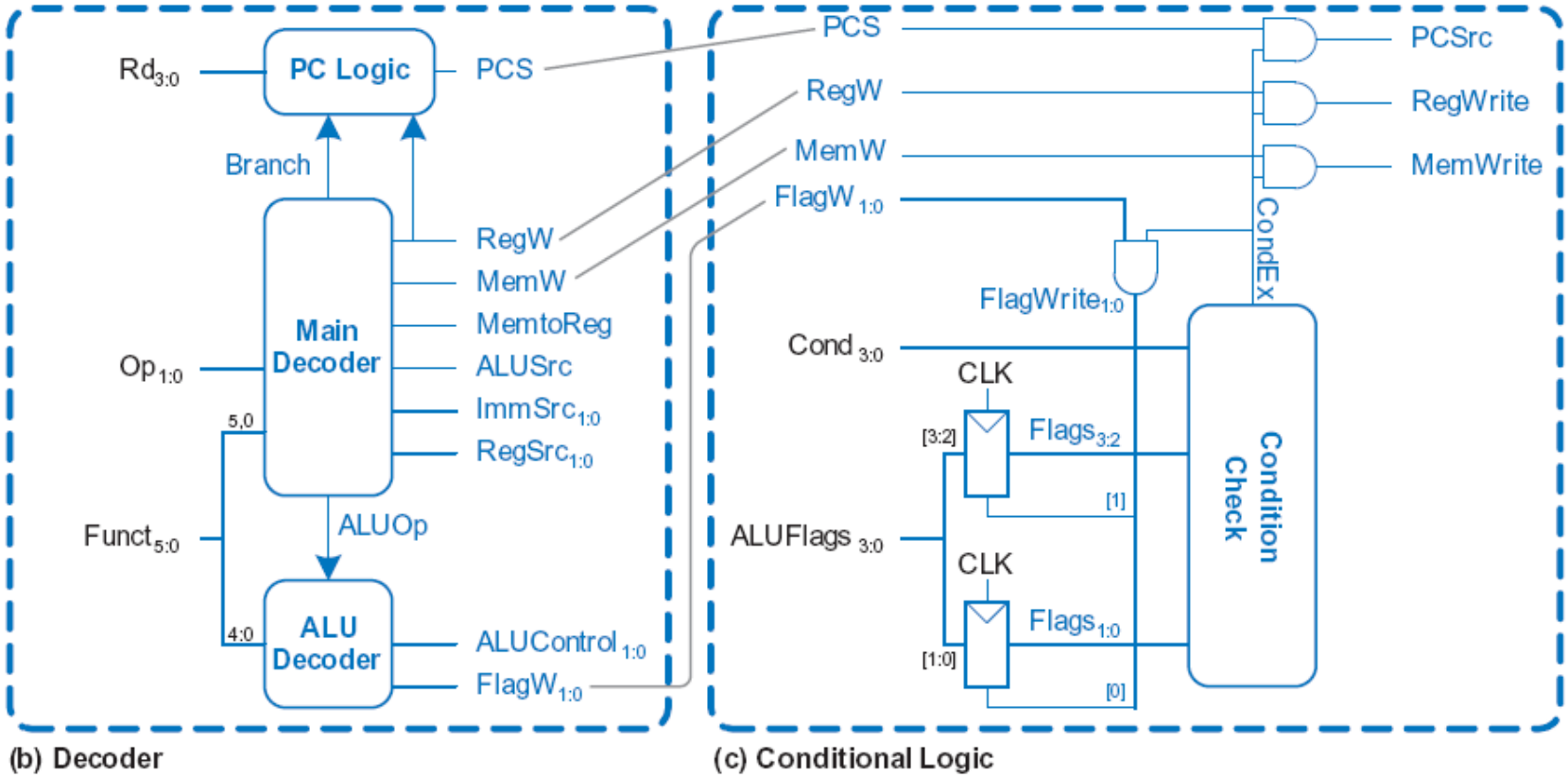


Se l'istruzione è eseguita: ***PCSrc*** **=** ***PCS***

Altrimenti: ***PCSrc*** **= 0 (e quindi, PC = PC + 4)**

Logica condizionale

I segnali che abilitano la scrittura (**MemW** and **RegW**) e l'aggiornamento dei flag (**FlagWrite**) e del PC (**PCS**) devono *passare* attraverso la logica condizionale prima di diventare operativi (e.g. segnali datapath **MemWrite**, **RegWrite** e **PCSrcWrite**). Tali segnali possono essere azzerati dalla logica condizionale, se la condizione non è soddisfatta.



Condizioni su flags di stato

Mnemonic	Name	CondEx
EQ	Equal	Z
NE	Not equal	$\bar{Z}$
CS / HS	Carry set / Unsigned higher or same	C
CC / LO	Carry clear / Unsigned lower	$\bar{C}$
MI	Minus / Negative	N
PL	Plus / Positive of zero	$\bar{N}$
VS	Overflow / Overflow set	V
VC	No overflow / Overflow clear	$\bar{V}$
HI	Unsigned higher	$\bar{Z}C$
LS	Unsigned lower or same	$Z\ OR\ \bar{C}$
GE	Signed greater than or equal	$\overline{N \oplus V}$
LT	Signed less than	$N \oplus V$
GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$
AL	Always / unconditional	ignored

Condizioni su flags di stato

Condizioni sul risultato di una operazione aritmetica in complemento a 2

Mnemonic	Name	CondEx
MI	Minus / Negative	N
PL	Plus / Positive of zero	$\bar{N}$
VS	Overflow / Overflow set	V
VC	No overflow / Overflow clear	$\bar{V}$
AL	Always / unconditional	ignored

Condizioni su flags di stato

Per confrontare due interi (signed o unsigned) A e B si esegue la differenza A-B e si verificano le seguenti condizioni sui flag

Mnemonic	Name	CondEx
EQ	Equal	Z
NE	Not equal	$\bar{Z}$
CS / HS	Carry set / Unsigned higher or same	C
CC / LO	Carry clear / Unsigned lower	$\bar{C}$
HI	Unsigned higher	$\bar{Z}C$
LS	Unsigned lower or same	$Z\ OR\ \bar{C}$
GE	Signed greater than or equal	$\overline{N \oplus V}$
LT	Signed less than	$N \oplus V$
GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$

Condizioni su flags di stato

Mnemonic	Name	CondEx
EQ	Equal	Z
NE	Not equal	$\bar{Z}$

A==B sse A-B==0 sse Z=1

A!=B sse A-B!=0 sse Z=0

Vale sia per la rappresentazione in complemento a 2 (interi con segno) che nella rappresentazione senza segno

Condizioni su flags di stato

Mnemonic	Name	CondEx
GE	Signed greater than or equal	$\overline{N \oplus V}$
LT	Signed less than	$N \oplus V$
GT	Signed greater than	$\bar{Z}(\overline{N \oplus V})$
LE	Signed less than or equal	$Z\ OR\ (N \oplus V)$

A<B sse A-B<0 sse (N $\oplus$ V)

- Quando A-B non genera overflow allora deve essere negativo
  - Questo accade sempre quando A e B hanno lo stesso segno
- Quando A-B genera overflow?
  - Allora A e B hanno segni diversi
  - A<B è vera sse A negativo e B positivo sse A negativo e (-B) negativo, e quindi, essendoci overflow il risultato deve essere positive (ovvero N=0)

A<=B sse A-B<0 or A-B==0 sse Z+(N $\oplus$ V)

Condizioni su flags di stato

Mnemonic	Name	CondEx
CS / HS	Carry set / Unsigned higher or same	C
CC / LO	Carry clear / Unsigned lower	$\bar{C}$
HI	Unsigned higher	$\bar{Z}C$
LS	Unsigned lower or same	$Z\ OR\ \bar{C}$

Problema A-B che semantica ha nella rappresentazione senza segno?

- Sappiamo che A-B è in complemento a due A+(~B+1), dove ~B è la negazione bit a bit di B
- Nella rappresentazione senza segno invece
  - $\sim(X_{N-1}\ldots X_0)= (1-X_{N-1})\ldots(1-X_0)=1\ldots1-(X_{N-1}\ldots X_0)=2^N-1-B$

1	1	1	1	1	1	1	1
1	0	0	0	1	1	1	0
0	1	1	1	0	0	0	1

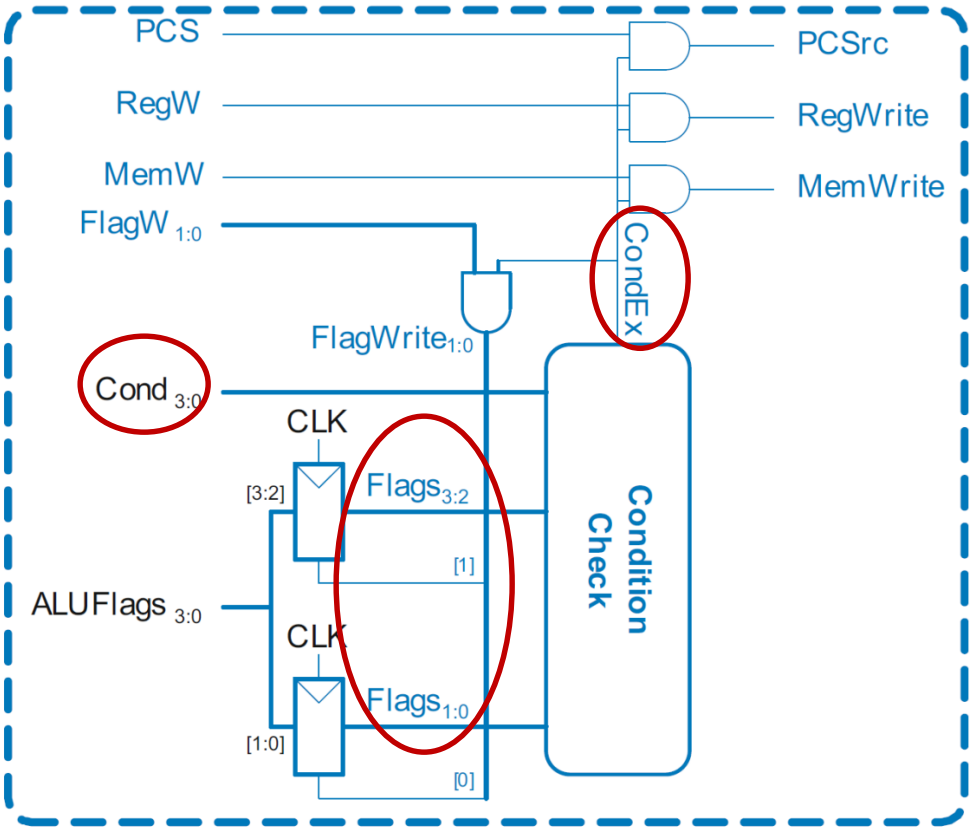
Condizioni su flags di stato

- Quindi A+(~B+1) corrisponde nella rappresentazione senza segno a 2^N+A-B
- Ora 2^N+A-B non genera un carry out sse 2^N+A-B<= 2^N-1 sse A+1<=B sse A<B
- Quindi A<B è verificato dalla condizione $\bar{C}$

- A<=B sse A<B o A==B sse $Z + \bar{C}$
- A>B sse !(A<=B) sse $\bar{Z}C$

Esempi

$Flags_{3:0}$ = NZCV



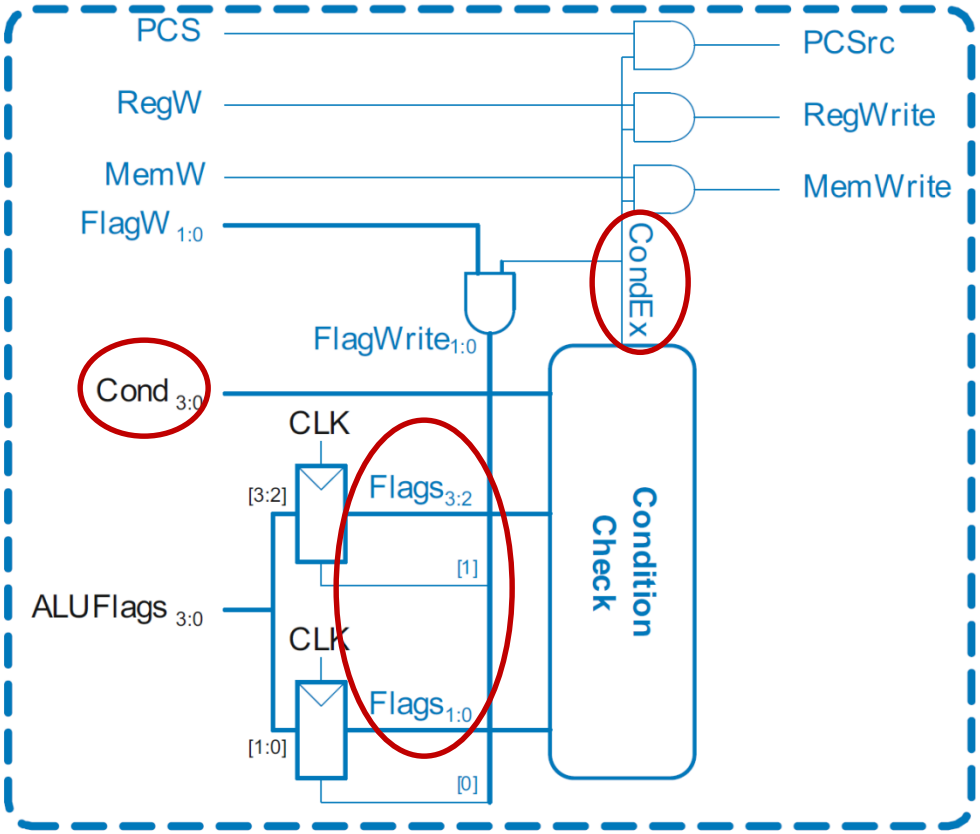

```
AND R1, R2, R3
```

$Cond_{3:0}$=1110 (istruzione non condizionata) =>

CondEx = 1

Esempi

Flags$_{3:0}$ = NZCV

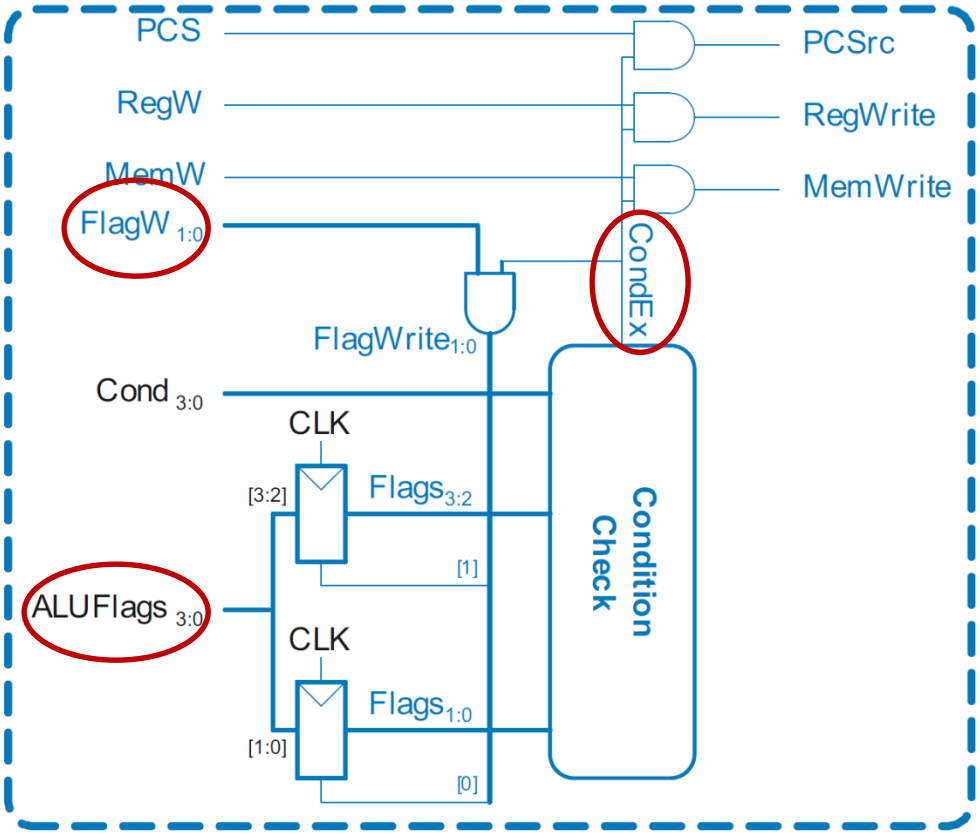



```
EOREQ R5, R6, R7
```

Cond$_{3:0}$=0000 (EQ): if **Flags$_{3:2}$**=X1XX => **CondEx** = 1

Update dei flags di stato

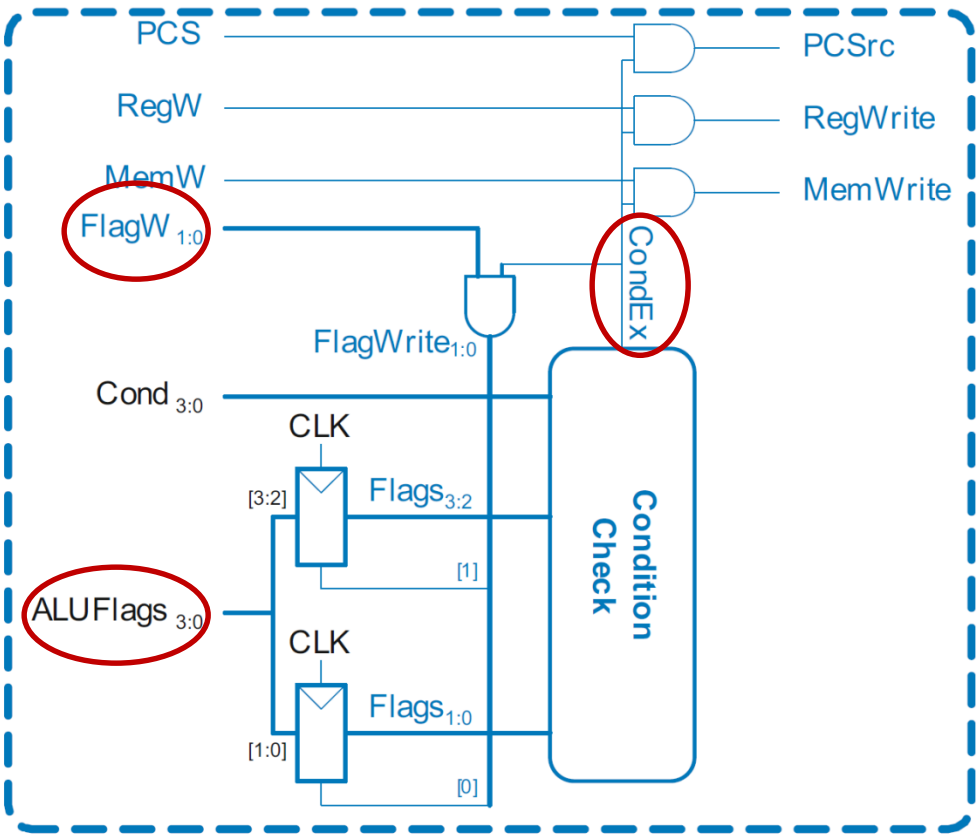
Flags$_{3:0}$ = NZCV



Flags$_{3:0}$ vengono aggiornati con i valori di ALUFlags$_{3:0}$ se si verificano le seguenti condizioni:

- **FlagW** è 1 (ovveri, S-bit dell'istruzione corrente è 1)
- **CondEx** è 1 (l'istruzione deve essere eseguita)

Esempi

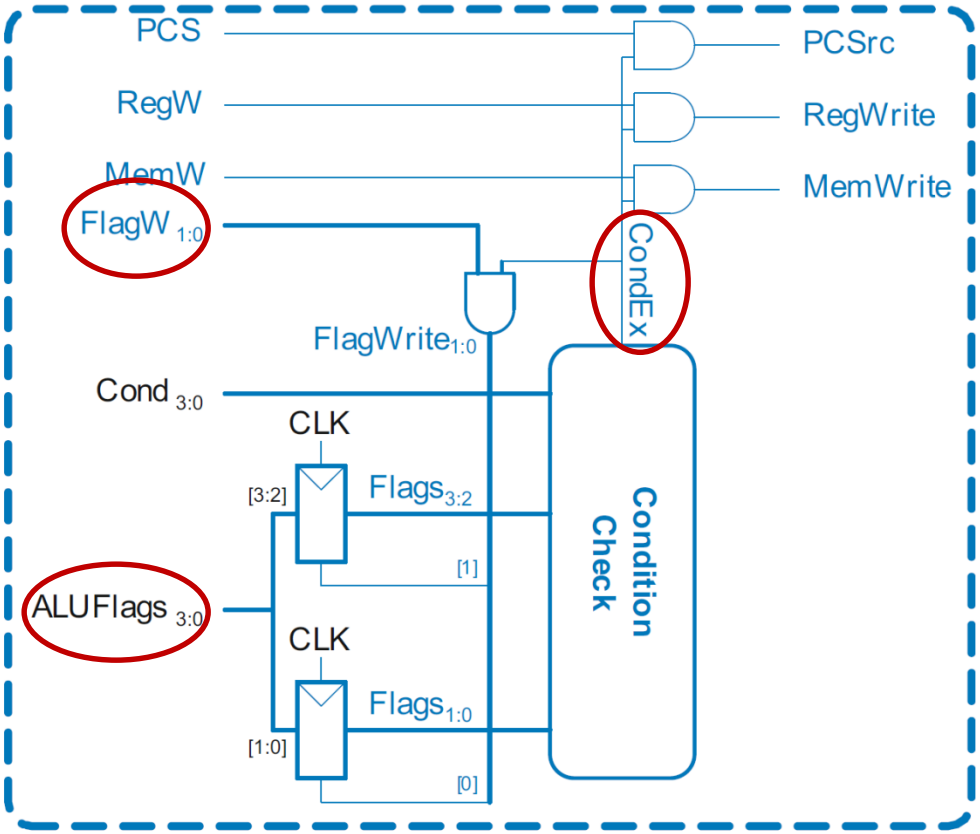



```
SUBS R5, R6, R7
```

$FlagW_{1:0}$ = 11 e **CondEx** = 1 (istruzione incondizionata) =>

$FlagWrite_{1:0}$ = 11 **Tutti i flag vengono aggiornati**

Esempi




```
ANDS R7, R1, R3
```

FlagW$_{1:0}$ = 10 E **CondEx** = 1 (istruzione incondizionata) =>

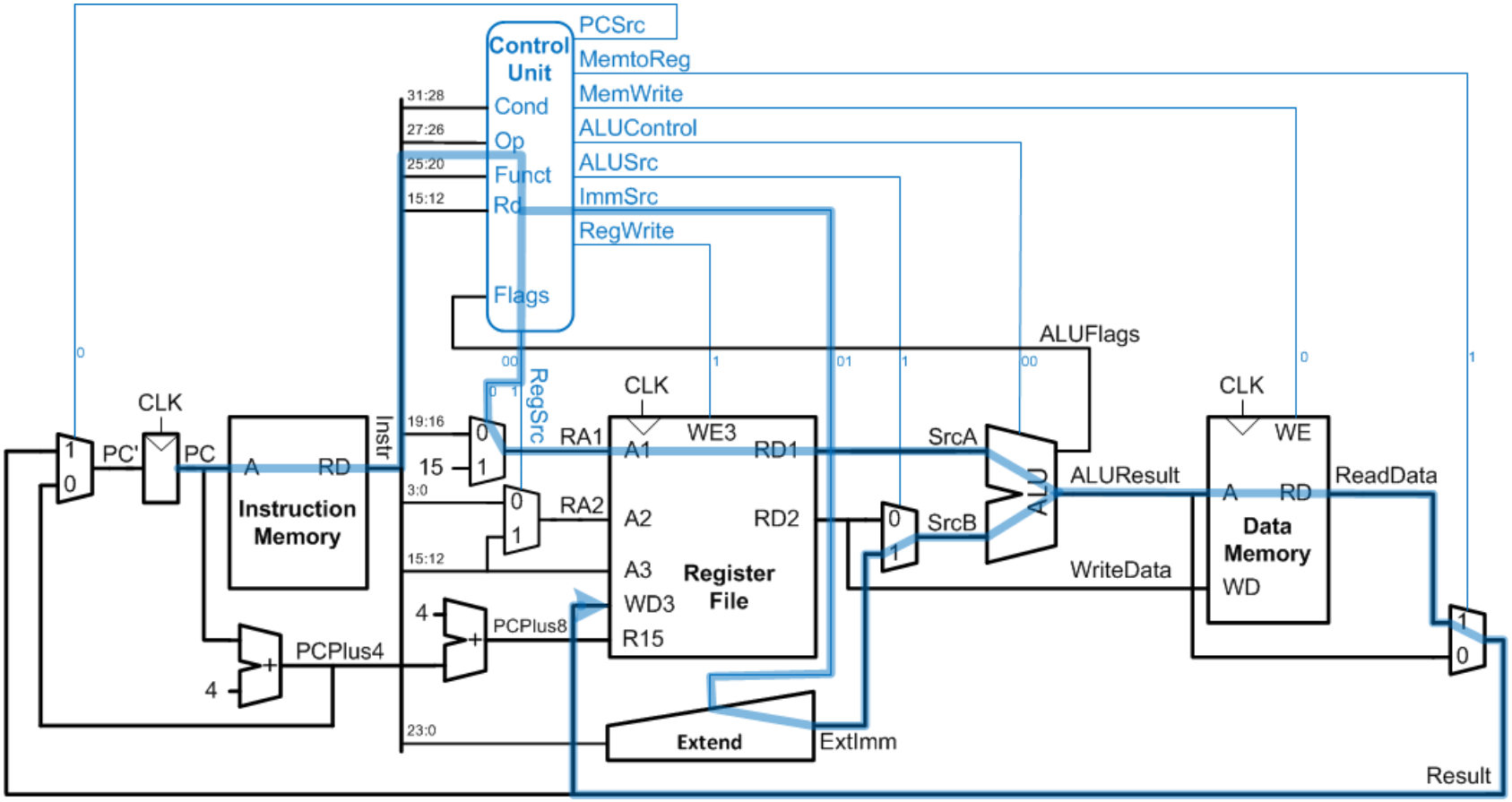
FlagWrite$_{1:0}$ = 10 Only **Flags$_{3:2}$** sono aggiornati

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.



T_c limited by critical path (`LDR`)

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.

Limiti del ciclo singolo

Le architetture a ciclo singolo hanno tre principali limiti:

- **separazione delle memorie**: la memoria istruzioni e la memoria dati devono essere necessariamente separate, poiché dati e istruzioni devono essere gestiti all'interno dello stesso ciclo.

- **inefficienza temporale**: il ciclo di clock deve avere una durata pari al tempo impiegato dall'istruzione più lenta, sprecando tempo per tutte quelle istruzioni molto più veloci.

- **duplicazione delle componenti**: lo stesso componente non può essere riutilizzato per scopi distinti; ad esempio sono necessarie tre ALU, due per la gestione del PC e una per l'esecuzione delle istruzioni.

Vantaggi del ciclo multiplo

Le architetture a ciclo multiplo risolvono tali problemi, partizionando una intera istruzione in più passi, ciascuno dei quali viene eseguito in un ciclo di clock differente.

- È possibile utilizzare una sola memoria comune sia per le istruzioni, che per i dati. Infatti, l'istruzione viene letta in un ciclo, mentre i dati vengono letti o scritti in memoria in un ciclo differente.

- Istruzioni meno complesse richiedono un minor numero di cicli di clock, evitando sprechi di tempo.

- È possibile utilizzare un'unica ALU sia per gestire il PC, che per eseguire le istruzioni, purché tali operazioni siano effettuate in cicli di clock differenti.

Datapath

Il **datapath** è sviluppato in modo incrementale, i nuovi collegamenti sono evidenziati in **nero** (o **blu**), mentre quanto già introdotto è rappresentato in **grigio**.

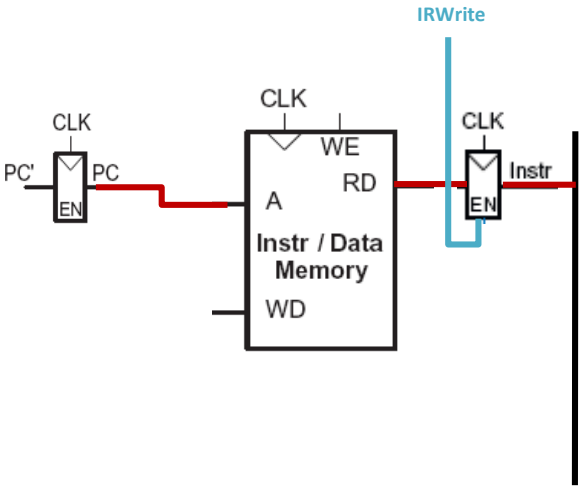
Al fine di facilitare la comprensione dell'architettura di un processore ARM, considereremo un limitato set di istruzioni:

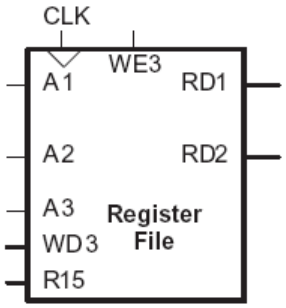
- le istruzioni di elaborazione dati: **ADD**, **SUB**, **AND**, **ORR** (con registro e modalità di indirizzamento diretto e senza shift);
- le istruzioni di Memoria: **LDR**, **STR** (diretto e con offset positivo);
- le istruzioni di salto (branch): **B**.

Datapath LDR

Il **PC** contiene l'indirizzo dell'istruzione da eseguire. Il primo passo è quello di leggere questa istruzione dalla memoria istruzioni, per cui il PC viene collegato all'indirizzo di ingresso della memoria.

L'istruzione a 32 bit viene letta e memorizzata in un registro **IR**. Il registro **IR** riceve un segnale **IRWrite** che indica quando caricare una istruzione.



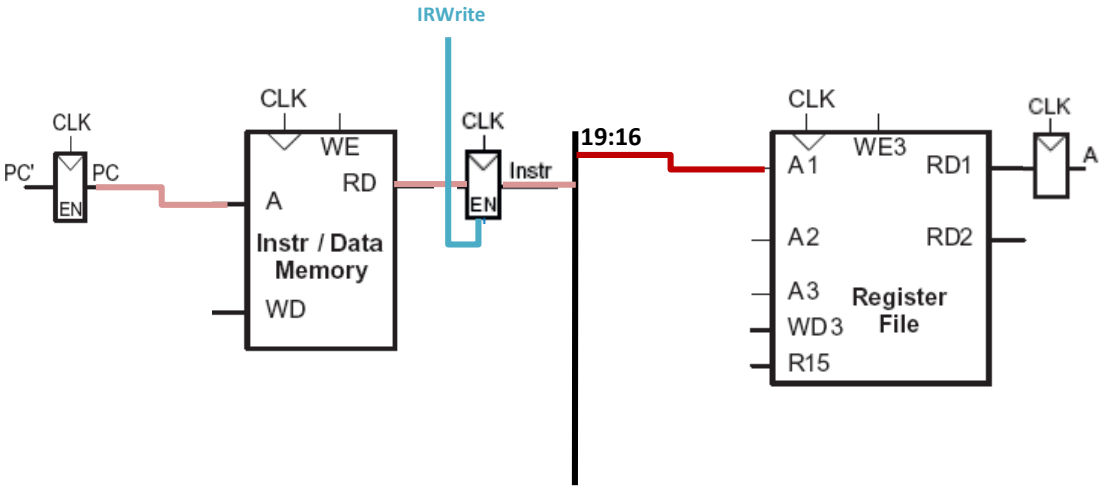


Datapath LDR

Il passo successivo è quello di leggere il registro sorgente contenente l'indirizzo di base. Questo registro è specificato nel campo **Rn** dell'istruzione, **$Instr_{19:16}$**

Questi bit vengono collegati all'ingresso indirizzo di una delle porte del file register (**A1**).

Il register file legge il valore di registro in **RD1** e lo memorizza in un registro **A**.

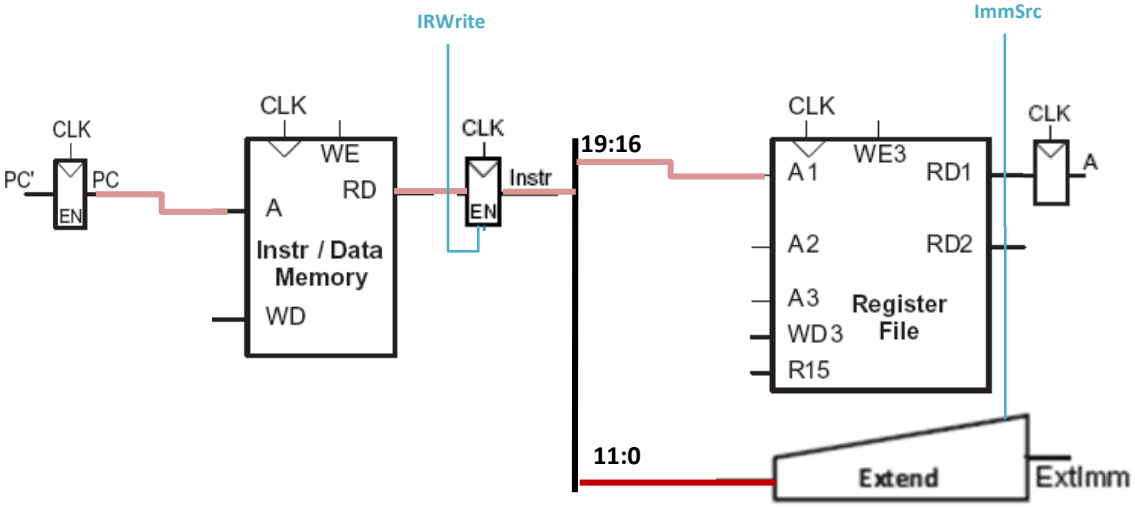


Datapath LDR

L'istruzione **LDR** richiede anche un **offset**, il quale è memorizzato nell'istruzione stessa e corrisponde ai bit $\mathbf{Instr}_{11:0}$.

L'offset è un valore senza segno, quindi deve essere esteso a 32 bit.

Il valore a 32 bit (**ExtImm**) è tale che $\mathbf{ExtImm}_{31:12} = 0$ e $\mathbf{ExtImm}_{11:0} = \mathbf{Instr}_{11:0}$.
ExtImm estende a 32 bit costanti a 8, 12 e 24 bit. Non viene memorizzato in un registro, poiché dipende solo da **Instr**, che non cambia durante l'esecuzione dell'istruzione.

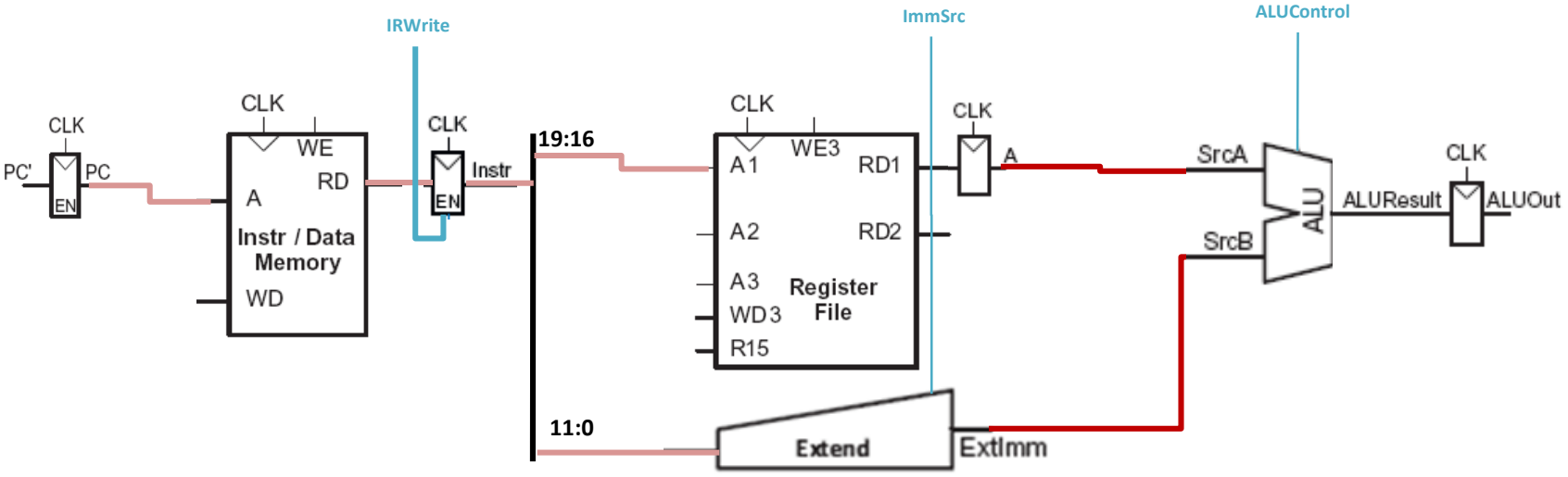


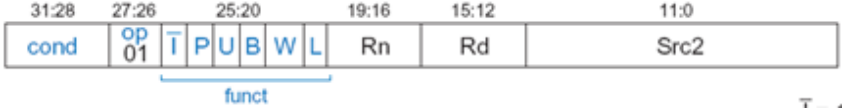
Datapath LDR

Il processore deve aggiungere l'**indirizzo di base** all'offset per trovare l'indirizzo di memoria a cui leggere. La somma è effettuata per mezzo di una **ALU**.

La **ALU** riceve due operandi (**srcA** e **srcB**). **srcA** proviene dal register file, mentre e **srcB** da ExtImm. Inoltre, il segnale a 2-bit **ALUControl** specifica l'operazione (00 per somma, 01 sottrazione).

La **ALU** genera un valore a 32 bit **ALUResult**, che viene memorizzato in un registro **ALUOut**.

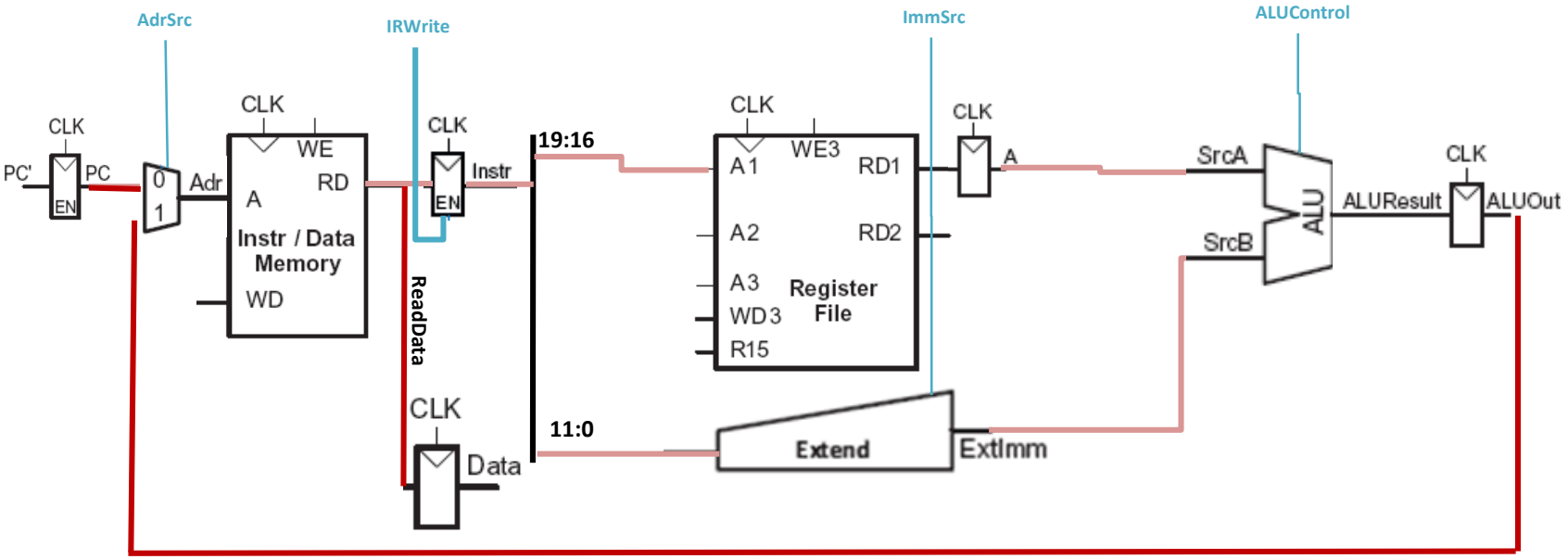




Datapath LDR

L'indirizzo calcolato dall'**ALU** deve essere inviato alla porta **A** della memoria. Serve un multiplexer per disambiguare l'accesso con **ALUOut** o **PC**. Il multiplexer è controllato dal segnale **AdrSrc**.

I dati vengono letti dalla memoria dati sul bus **ReadData**, e poi vengono memorizzati in un registro chiamato **Data**.

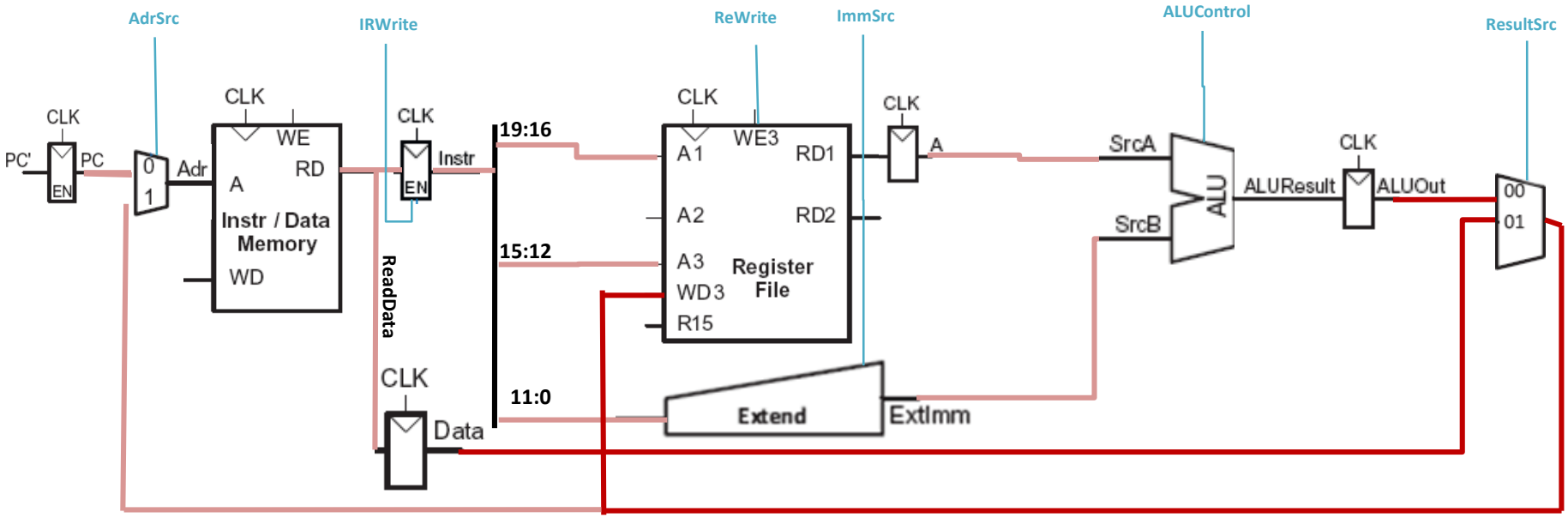


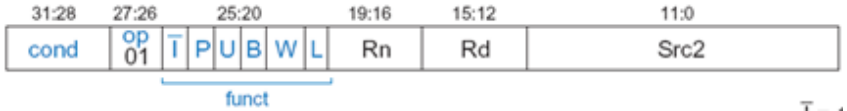
Datapath LDR

Inoltre, i dati appena letti devono essere scritti nel registro **Rd** specificato dai bit **15:12** dell'istruzione **Instr**.

Piuttosto che collegare direttamente **Data** alla porta **WD3**,si consideri che anche il risultato dell'**ALU** potrebbe dover essere scritto in **Rd**. Quindi aggiungiamo un multiplexer che seleziona fra **ALUOut** e **Data**.

Il segnale **RegWrite** deve essere impostato a **1**, per permettere la scrittura nel registro.

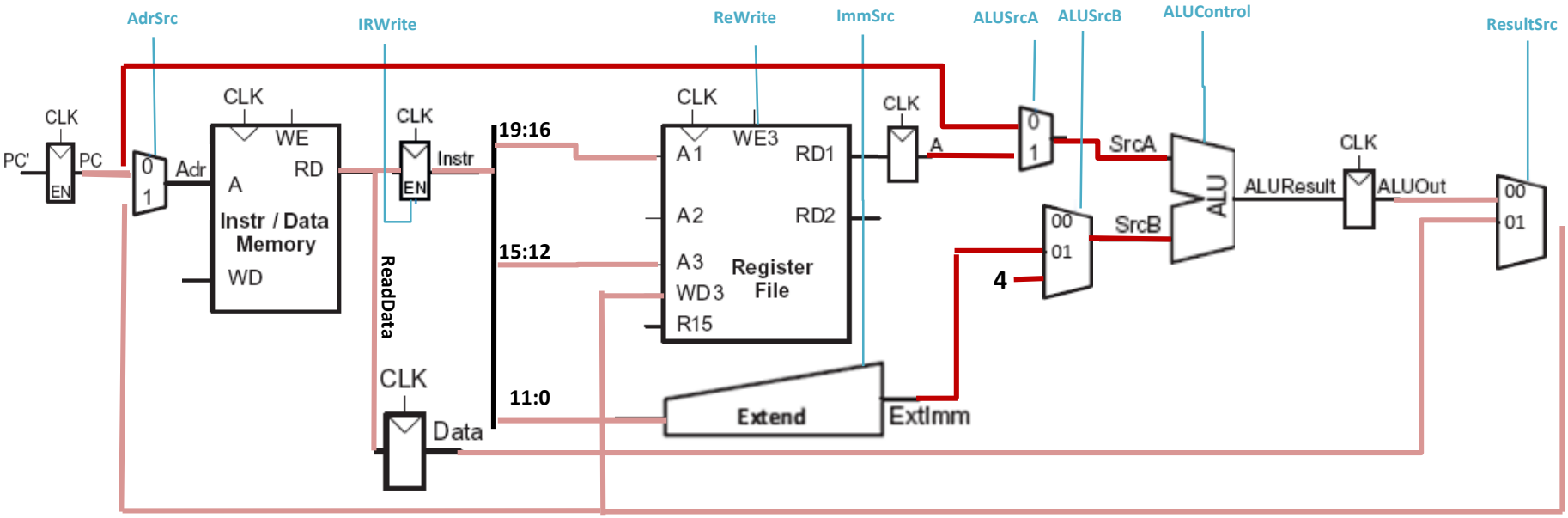


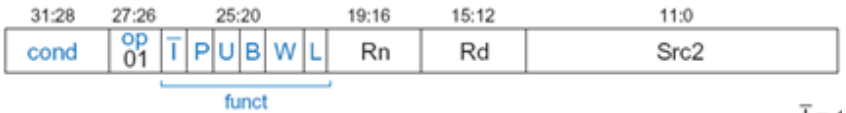


Datapath LDR

Un ulteriore compito a carico dell'**ALU** è l'incremento del **PC**, operazione che prima era svolta da una **ALU** diversa.

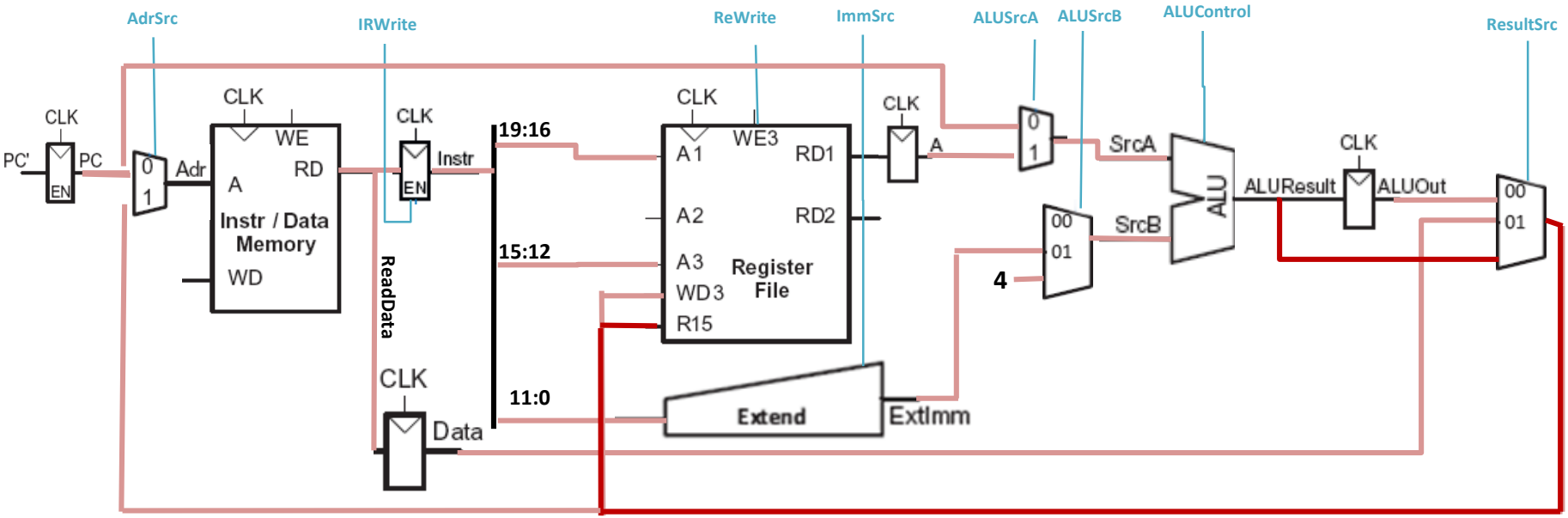
Aggiungiamo un multiplexer sul primo ingresso dell'**ALU**, che permette di scegliere fra il contenuto del registro **A** e il **PC**.
Sul secondo ingresso dell'ALU aggiungiamo un ulteriore multiplexer che permetta di selezionare fra ExtImm e la costante 4.

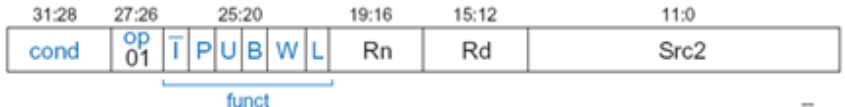




Datapath LDR

Si consideri infine, che il contenuto del registro **R15** nelle architetture ARM corrisponde a **PC+8**.
Durante il passo di fetch, il **PC** è stato aggiornato a **PC+4**, per cui sommare 4 al nuovo contenuto di **PC** produce **PC+8**, che viene memorizzato in **R15**.
Scrivere **PC+8** in **R15**, richiede che il risultato dell'ALU possa essere collegato a tale registro. A tal fine, colleghiamo **ALUResult** con uno dei tre ingressi del multiplexer.

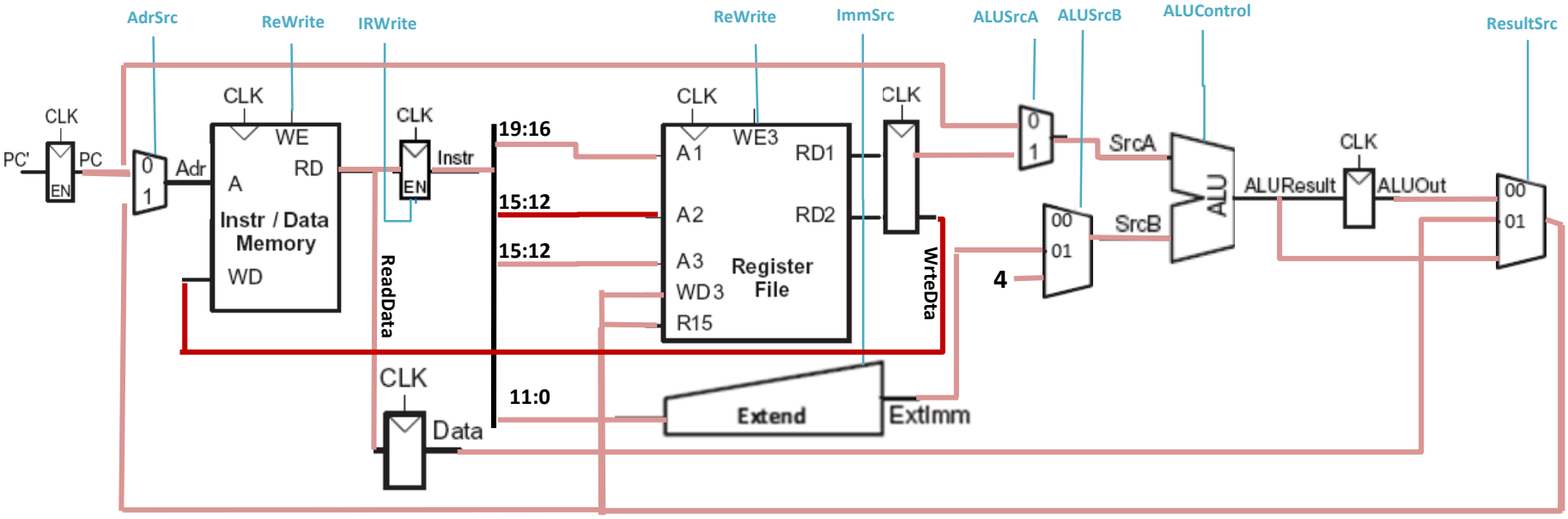




Datapath STR

Analogamente all'istruzione di caricamento, **STR** legge l'indirizzo di base dalla porta **RD1** del register file, estende la costante e l'**ALU** somma i due valori per calcolare l'indirizzo di memoria. Tutte queste operazioni sono già supportate.

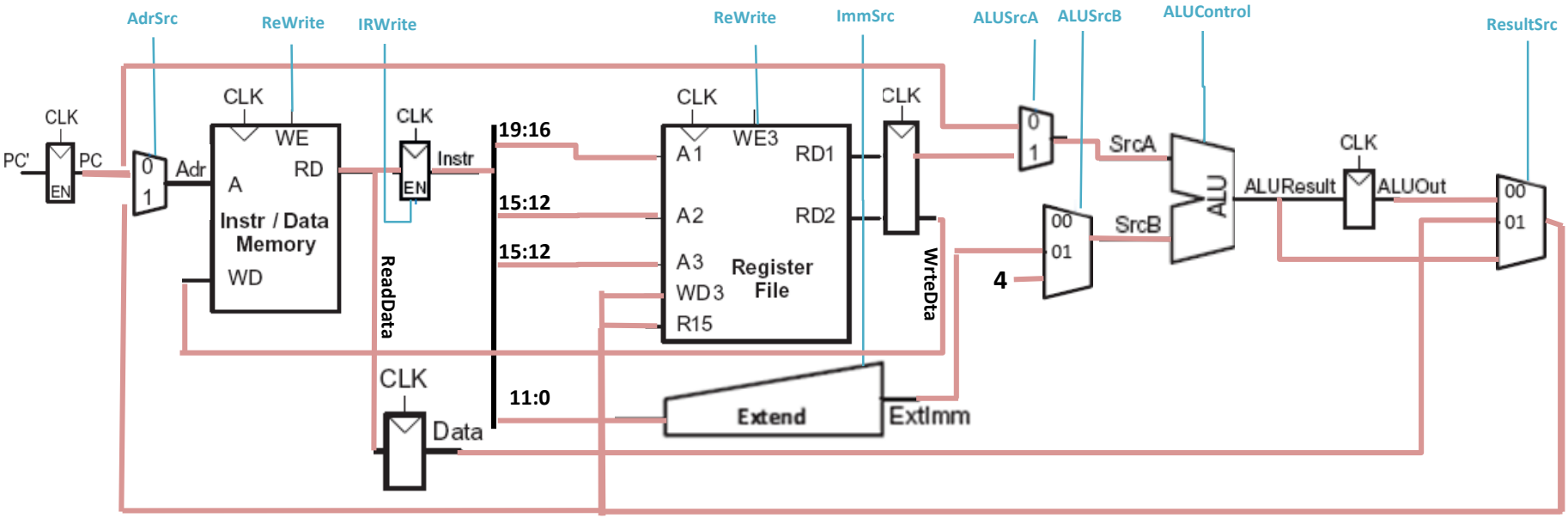
In aggiunta, **STR** legge il registro **Rd** da cui scrivere, che è specificato nei bit **$Instr_{15:12}$**. Il contenuto di **RD2** è inserito in un registro temporaneo WriteData e al passo successivo è inviato alla memoria, con segnale **MemWrite** attivo.



Datapath DataProcessing

Per le istruzioni di data processing con costante (**ADD**, **SUB**, **AND**, **OR**), il datapath legge il primo operando specificato da **Rn**, estende la costante da **8** a **32** bit, esegue l'operazione mediante l'**ALU** e scrive il risultato in un registro del register file. Tutte queste operazioni sono già supportate dal datapath.

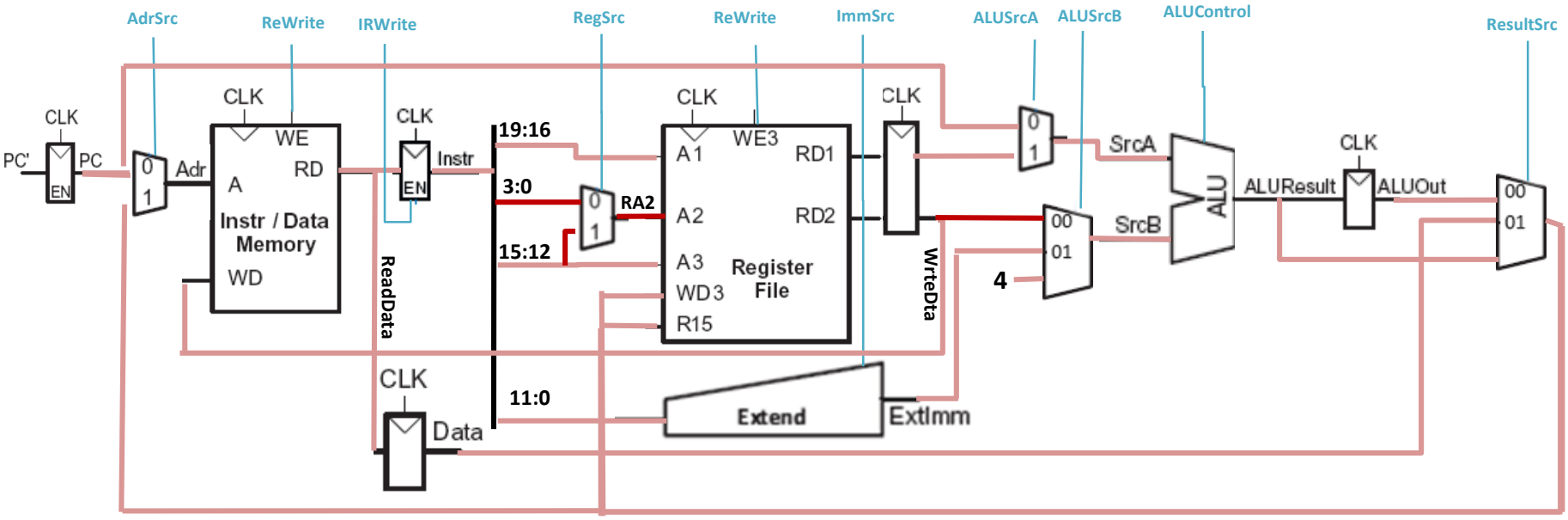
L'operazione da effettuare è specificata dal segnale **ALUControl**, mentre gli **ALUFlags** permettono di aggiornare il registro di stato.



Datapath DataProcessing

Per le istruzioni di data processing con registro (**ADD**, **SUB**, **AND**, **OR**), il datapath legge il secondo operando specificato da **Rm**, indicato nei bit $\mathbf{Instr_{3:0}}$.

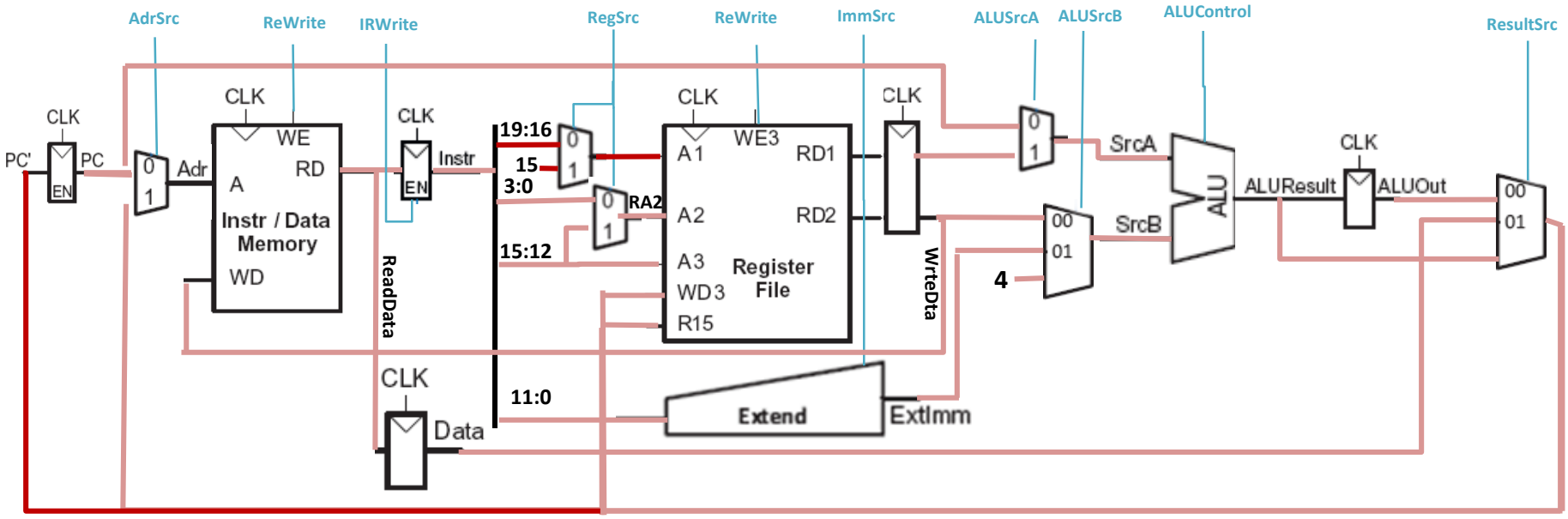
Inseriamo un multiplexer per selezionare tale campo sulla porta **A2** del register file. Il mutiplexer è controllato dal segnale **RegSrc**. Inoltre, estendiamo il multiplexer in **SrcB** in modo da considerare questo caso.



Datapath Branch

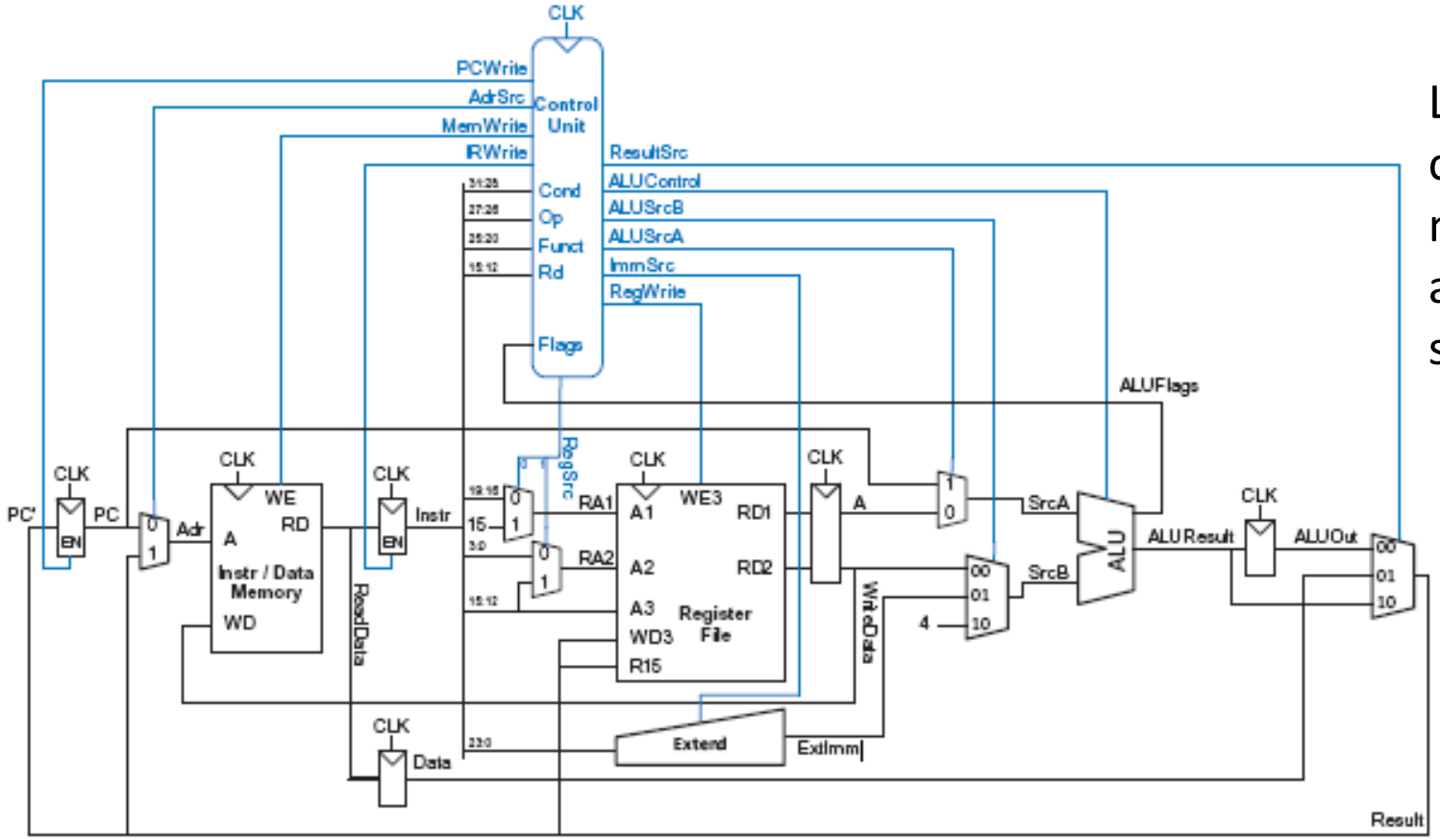
Per le istruzioni di branch, il datapath legge **PC+8** e una costante a **24** bit, che viene estesa a **32** bit. La somma di questi due valori è addizionata al **PC**.
Si ricorda, inoltre, che il registro **R15** contiene il valore **PC+8** e deve essere letto per tornare da un salto. È sufficiente aggiungere un multiplexer per selezionare **R15** come input sulla porta **A1**.

Il multiplexer è controllato dal segnale **RegSrc**.



Unità di controllo

Come nel processore a ciclo singolo, l'unità di controllo genera i segnali di controllo in base ai campi **cond**, **op** e **funct** dell'istruzione (**$Instr_{31:28}$**, **$Instr_{27:26}$**, e **$Instr_{25:20}$**, ai flag e al fatto che il registro destinazione sia o meno il PC.

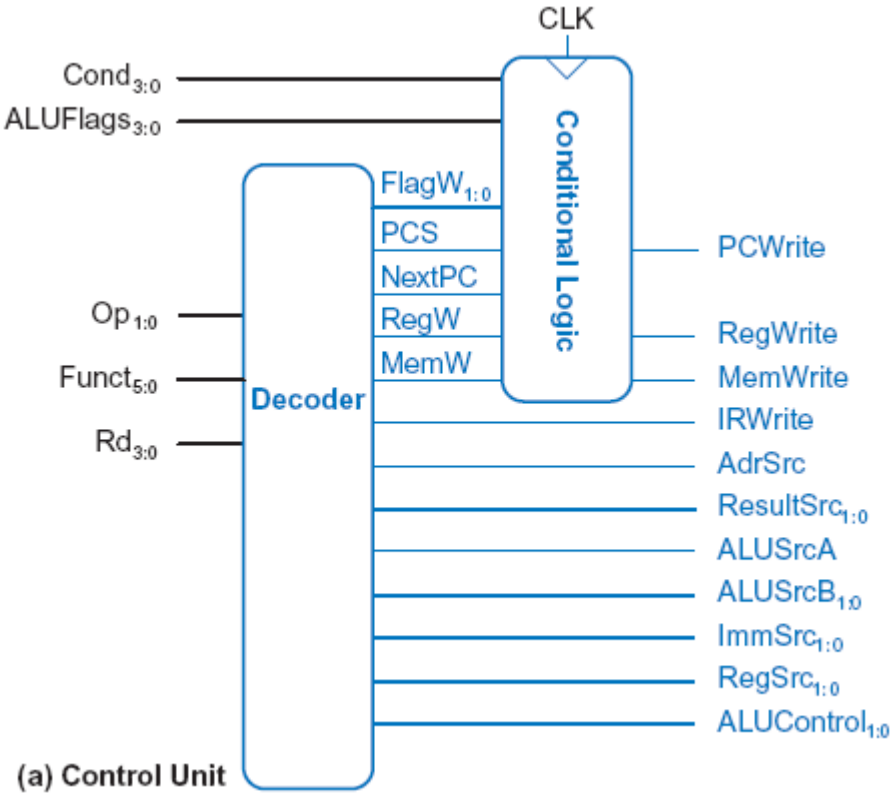


L'unità di controllo memorizza e aggiorna i flag di stato.

Unità di controllo

Come nel processore a ciclo singolo, l'unità di controllo è suddivisa in Decodificatore e logica condizionale. Il decodificatore è progettato come una FSM, che produce i segnali appropriati per i diversi cicli, sulla base del proprio stato.

Il decodificatore è realizzato con una macchina di Moore, in tal modo le uscite dipendono solo dello stato attuale.

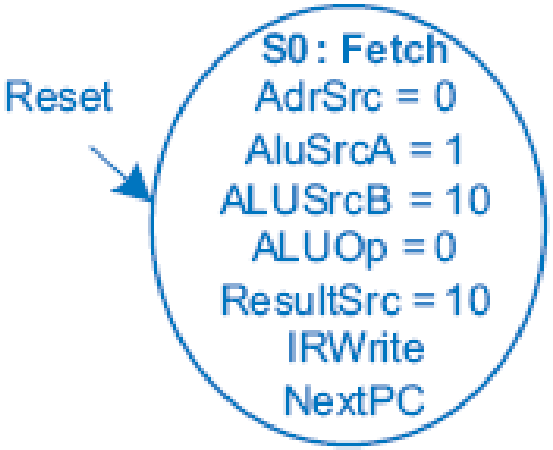


Dataflow di una istruzione

L'unità di controllo produce i segnali di attivazione per tutto il datapath (selezione nei multiplexer, abilitazione dei registri e scrittura in memoria).

Uno stato dell'automa che implementa il **main decoder** non è altro che lo stato dei segnali in un determinato momento.

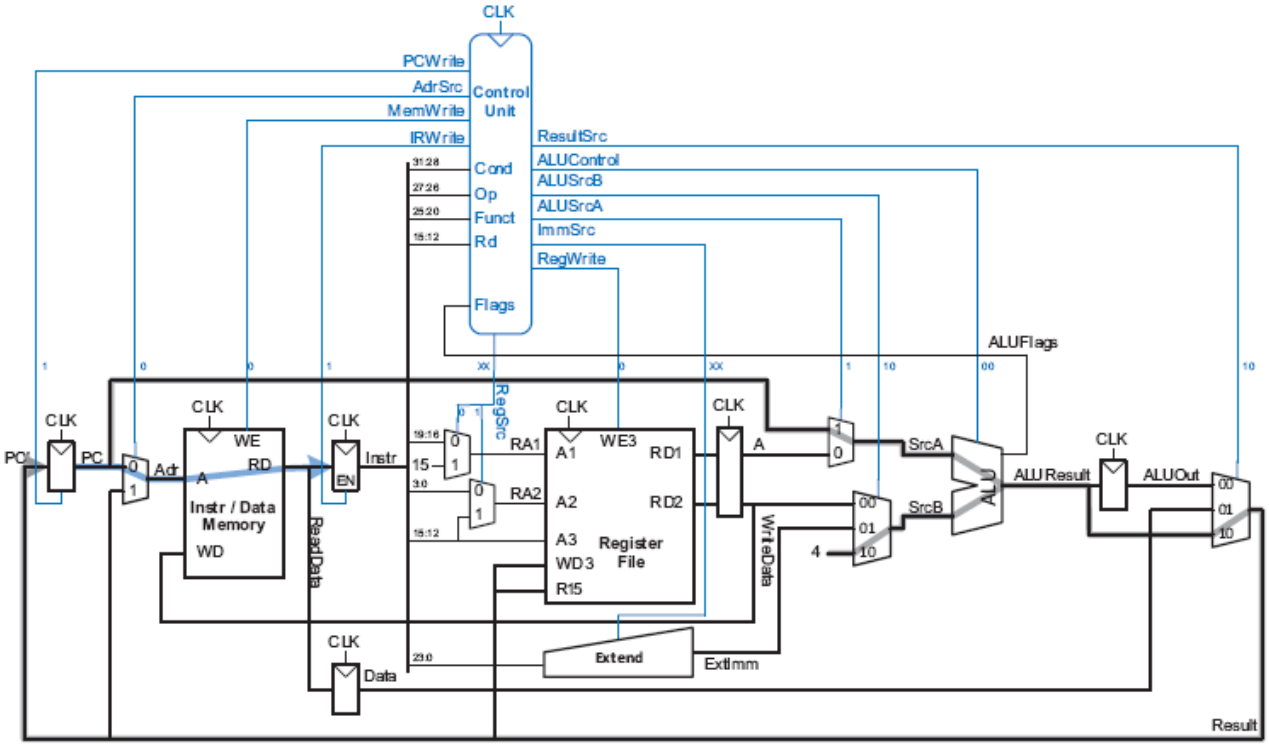
Per avere una visione più chiara di quali siano gli stati e di come vengano effettuate le transizioni da uno stato all'altro è utile considerare il **data flow** del processore.



In altri termini, data una istruzione osserviamo il comportamento del processore nei diversi cicli, in cui avvengono i quattro passi **fetch**, **decode**, **execute** e **store**.

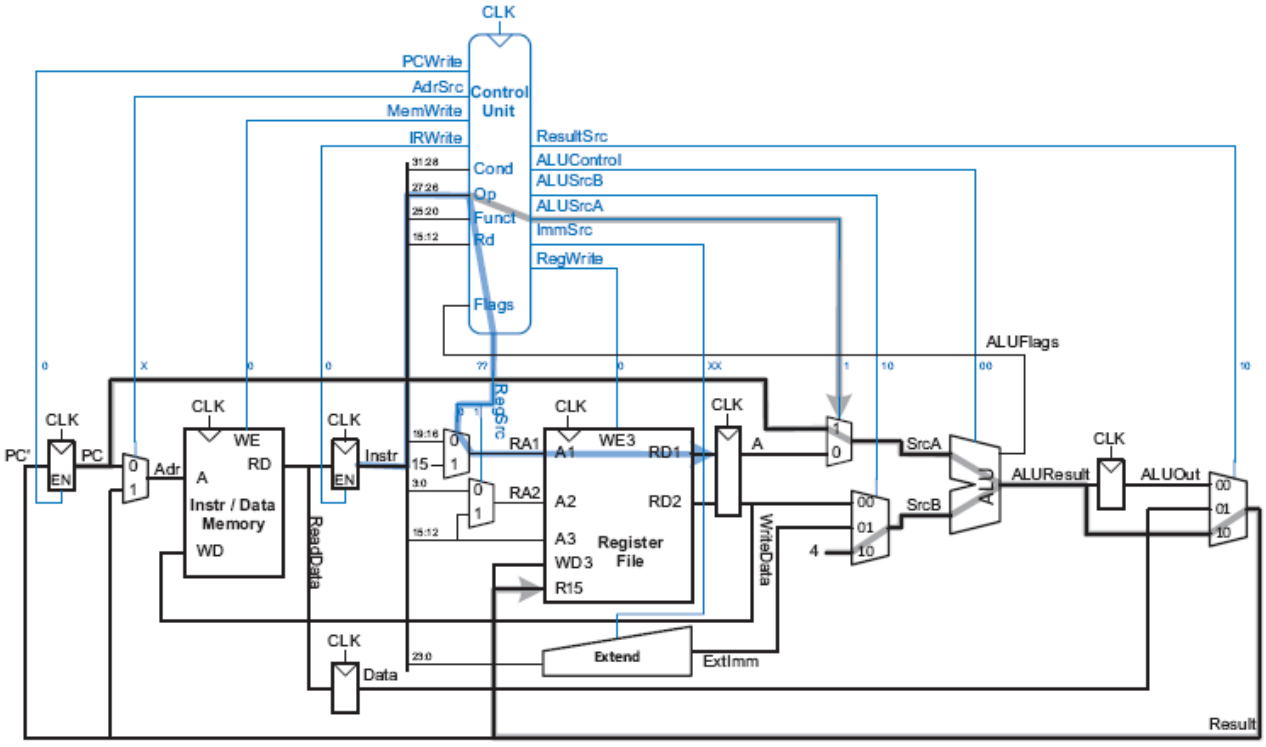
Dataflow di LDR

Operazione: **Fetch**



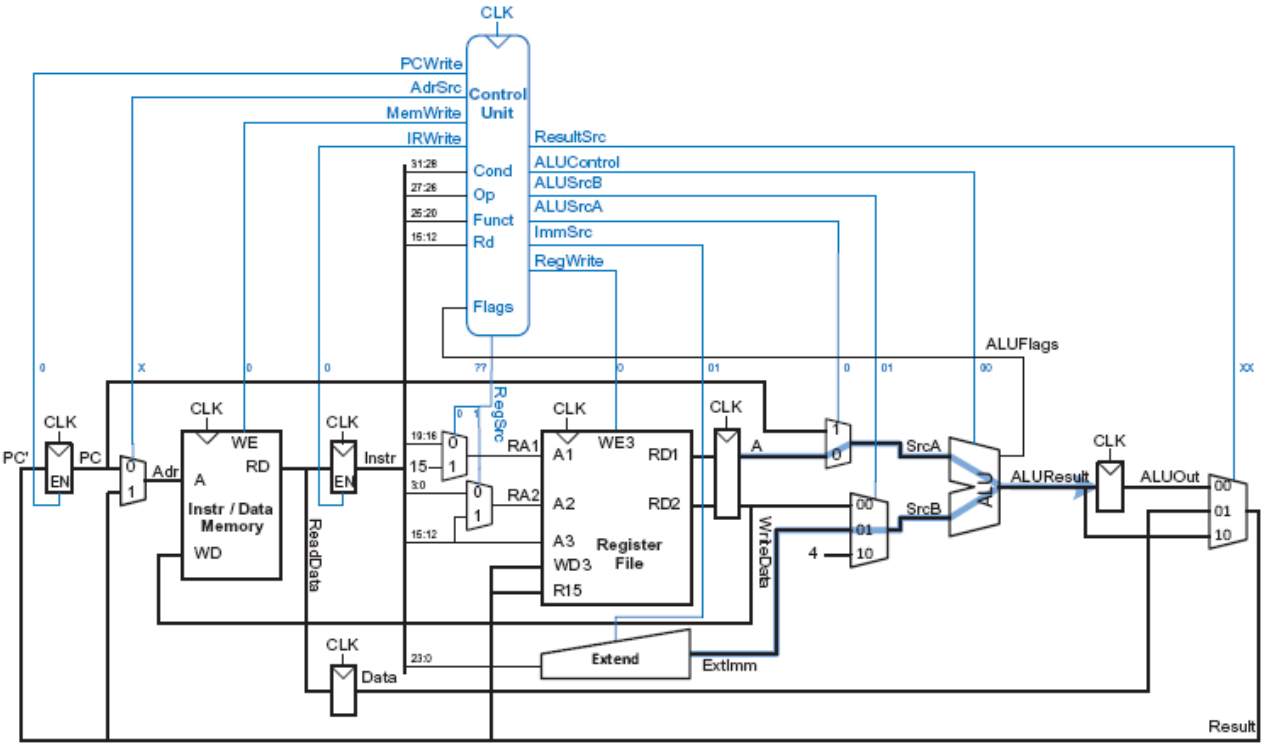
Dataflow di LDR

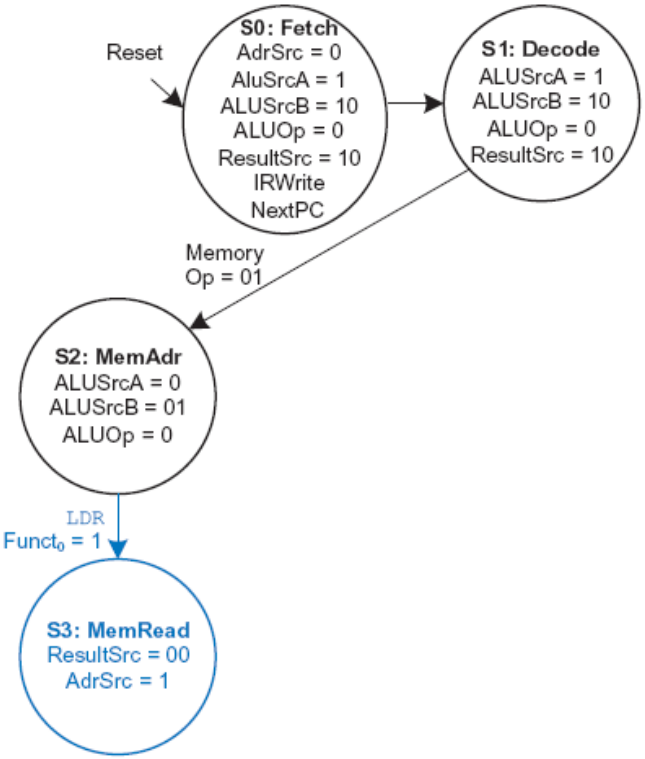
Operazione: **Decode**



Dataflow di LDR

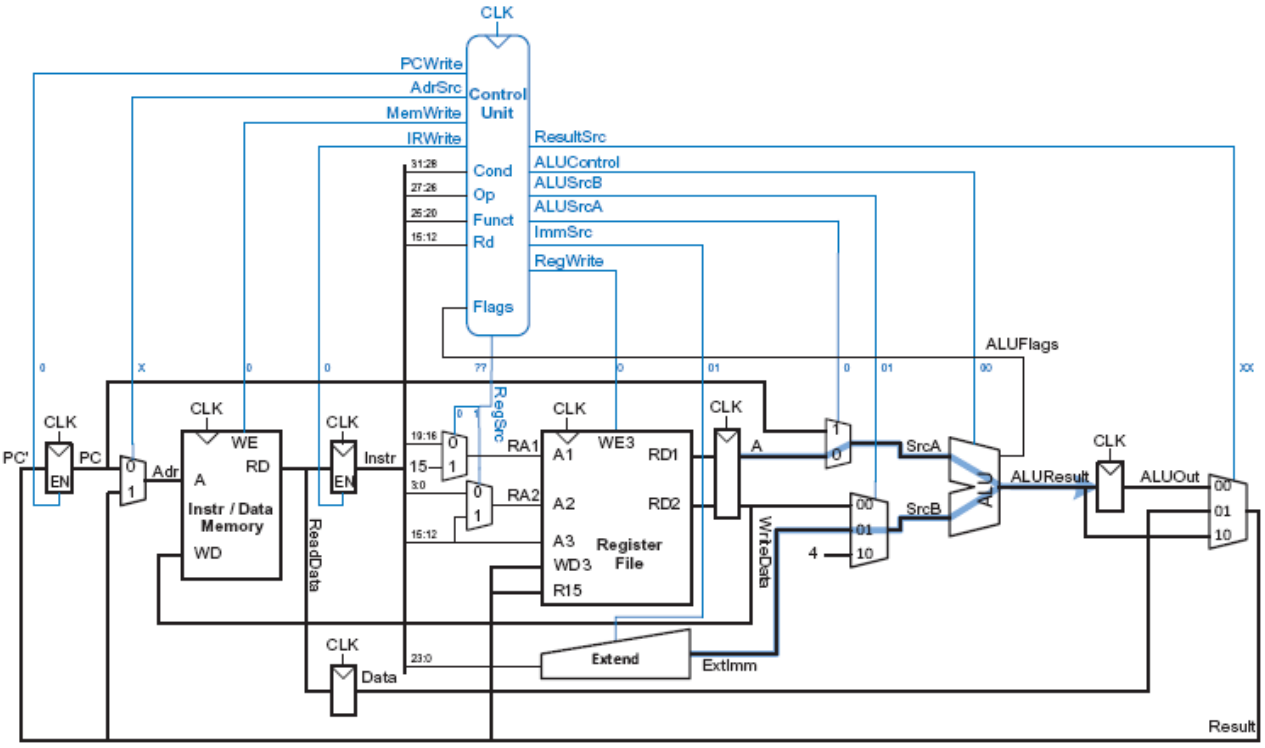
Operazione: **Execute** (memory address computation)

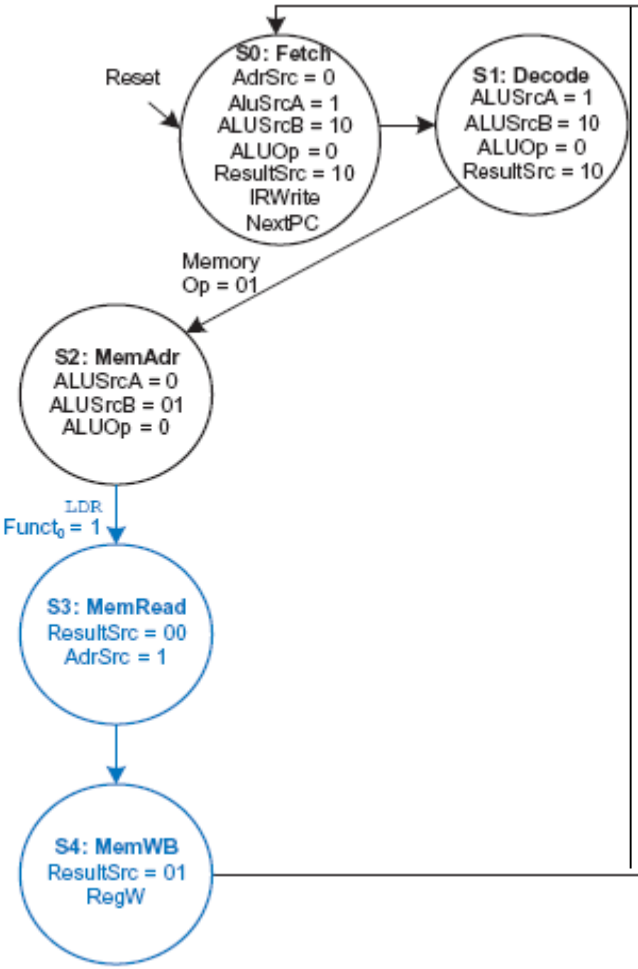




Dataflow di LDR

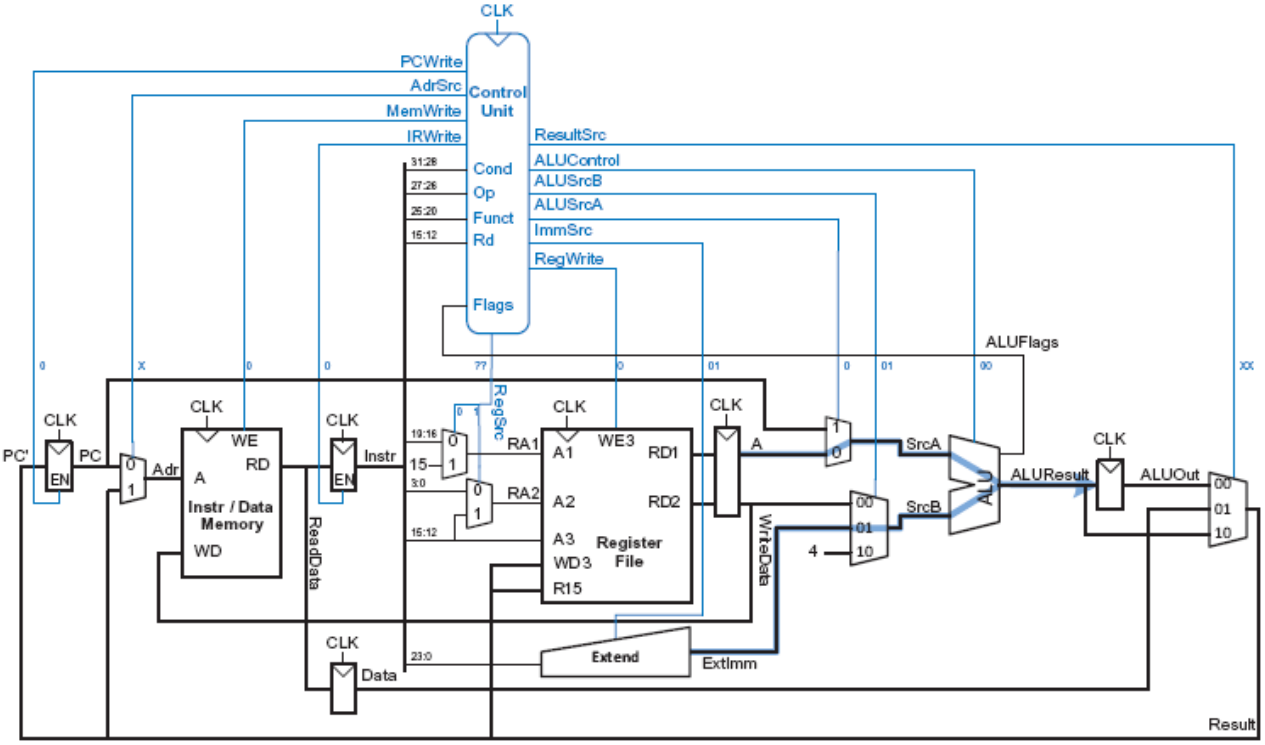
Operazione: **Store** (memory read)

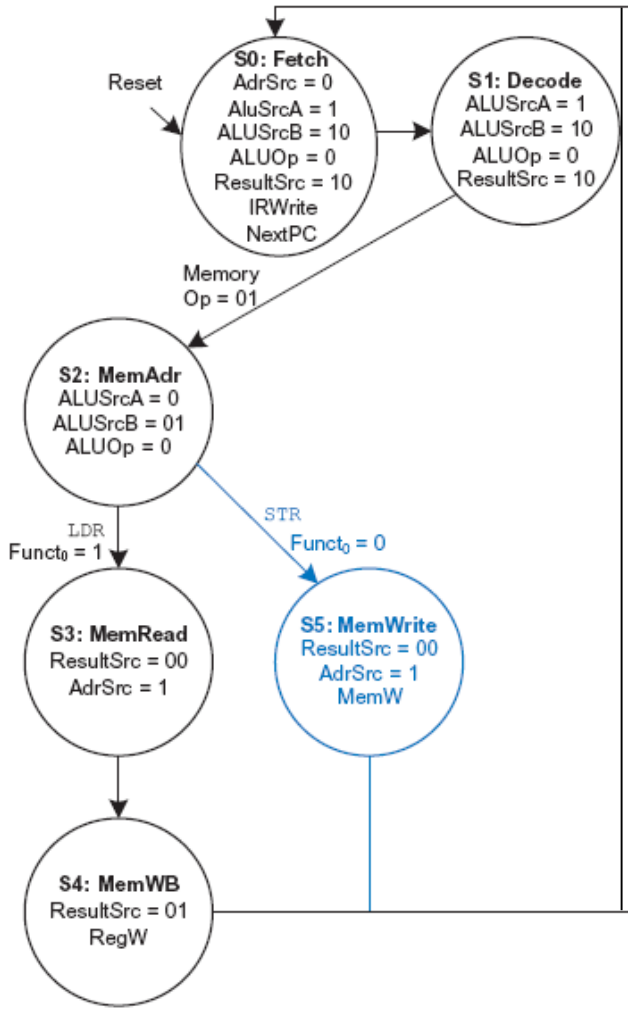




Dataflow di STR

Operazione: **Execute** (memory write)





Analisi delle prestazioni

In un processore a ciclo multiplo, il tempo impiegato per eseguire una istruzione dipende dal numero di cicli di clock, di cui necessita e dalla durata di un singolo ciclo di clock.

Il numero di cicli di clock necessario ad eseguire le diverse istruzioni è di:

- **branch** – **3** cicli;
- **data processing** – **4** cicli;
- **memory store** – **4** cicli;
- **memory load** – **5** cicli;

Valutiamo le prestazioni del processore a ciclo multiplo rispetto al benchmark SPECINT2000, che prevede:

- **LDR** – **25**%;
- **STR** – **10**%;
- **B** – **13**%;
- **Data Processing** – **52**%;

Analisi delle prestazioni

Il numero medio di cicli per istruzione è dato da:

CPI = (**perc. di B**)(**#cicli B**) + (**perc. di DP** + **perc. di STR**)(**#cili DP e STR**) + (**perc. di LDR**)(**#cili LDR**) =

= (**0.13**) (**3**) + (**0.52** + **0.10**) (**4**) + (**0.25**) (**5**) = **4.12**

I percorsi critici nel datapath, che richiedono maggior tempo e che quindi sono predominanti, sono due:

- Dal **PC**, attraverso il multiplexer **SrcA**, attraverso l'ALU, attraverso il multiplexer **Result**, attraverso la porta **R15**, fino al regitro **A**.

- Da **ALUOut**, attraverso il registro **Result**, attraverso il multiplexer **Adr**, attraverso la memoria (read), fino al registro **Data**.

- (t_{pcq_PC}) – caricamento di un nuovo indirizzo (PC) sul fronte di salita del clock;
- (t_{mux}) – selezione di un output da parte del multiplexer.
- (t_{ALU}) – l'ALU esegue su srcA e srcB una operazione.
- (t_{setup}) – viene impostato un segnale.

$$T_{c2} = t_{pcq\_PC} + 2t_{mux} + \max[t_{ALU} + t_{mux}, t_{mem}] + t_{setup};$$

Analisi delle prestazioni

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

Risposta:
secondo l'equazione

$T_{c2} = t_{pcq\_PC} + 2t_{mux} + \max[t_{ALU} + t_{mux}, t_{mem}] + t_{setup}$;

il tempo di ciclo del processore a ciclo multiplo è

Tc2 = 40 + 2(25) + 200 + 50 = 340 ps.

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) (4.12 cicli / istruzione) (340 × 10^{-12} s / ciclo) = 140 secondi.

Nota: **il tempo impiegato dal processore a ciclo singolo per lo stesso benchmark era di 84 sec.**

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

Considerazioni finali

Una delle motivazioni alla base della progettazione di un processore a ciclo multiplo è stata quella di evitare che il ciclo durasse quanto quello necessario all'istruzione più lenta.

Questo esempio dimostra che il processore a ciclo multiplo è più lento di quello a ciclo singolo a causa delle latenze di propagazione.

Infatti, sebbene l'istruzione più lenta (LDR) sia stata suddivisa in cinque fasi, il tempo di ciclo del processore non è aumentato di cinque volte.

Questo in parte perché:

- non tutti i passaggi hanno esattamente la stessa lunghezza;
- i tempi di setup sono necessari ad ogni passo, e non solo la prima volta per l'intera istruzione.

In generale, ci si è resi conto che il fatto che alcuni calcoli siano più veloci di altri è difficile da sfruttare, a meno che le differenze sono grandi.