ARCHITETTURA DEGLI ELABORATORI

A.A. 2020-2021

Università di Napoli Federico II
Corso di Laurea in Informatica

Docenti
Proff. Luigi Sauro gruppo 1 (A-G)
Silvia Rossi gruppo 2 (H-Z)

MICROARCHITETTURA ARM

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.

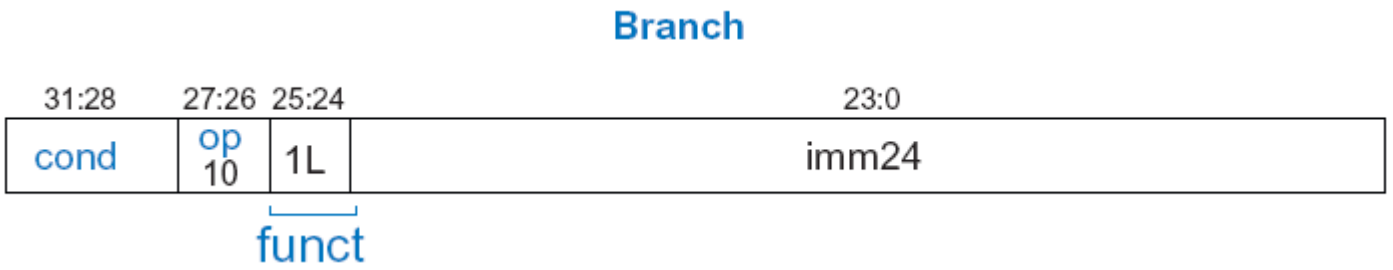
Istruzioni di Branching

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**).

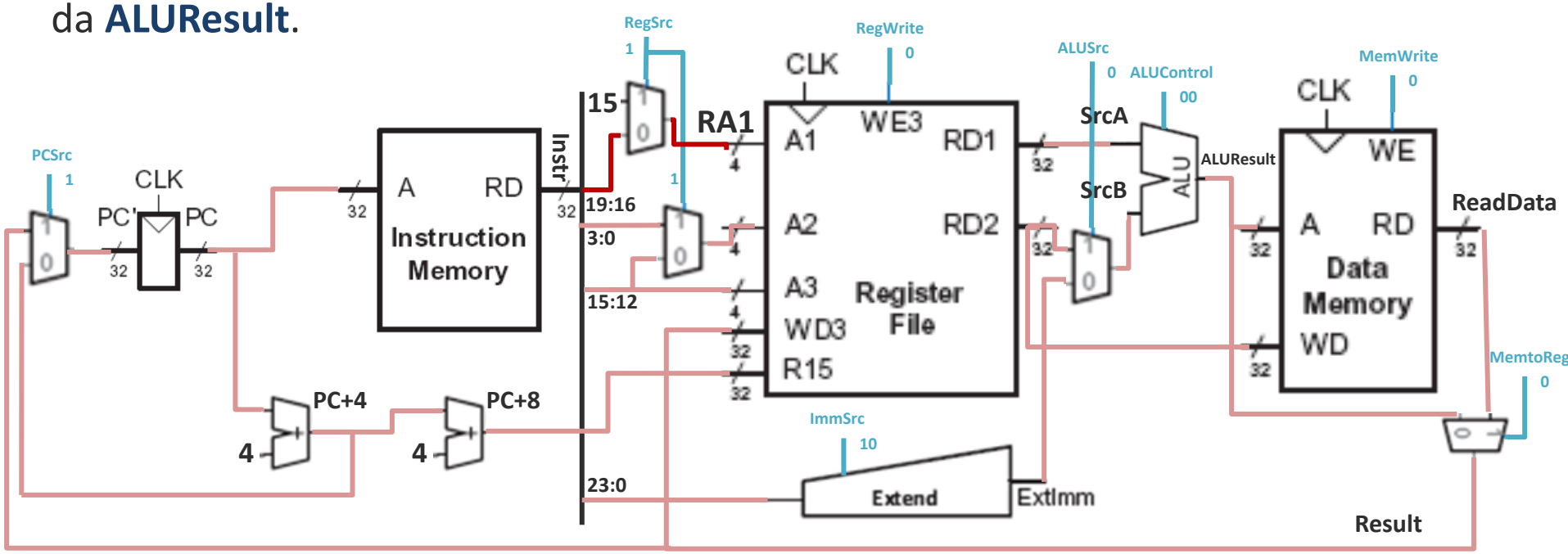
In esso viene salvato l'indirizzo di ritorno (ovvero il contenuto del registro **R15**) quando viene eseguita l'istruzione **BL** (Branch and Link).

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

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**.

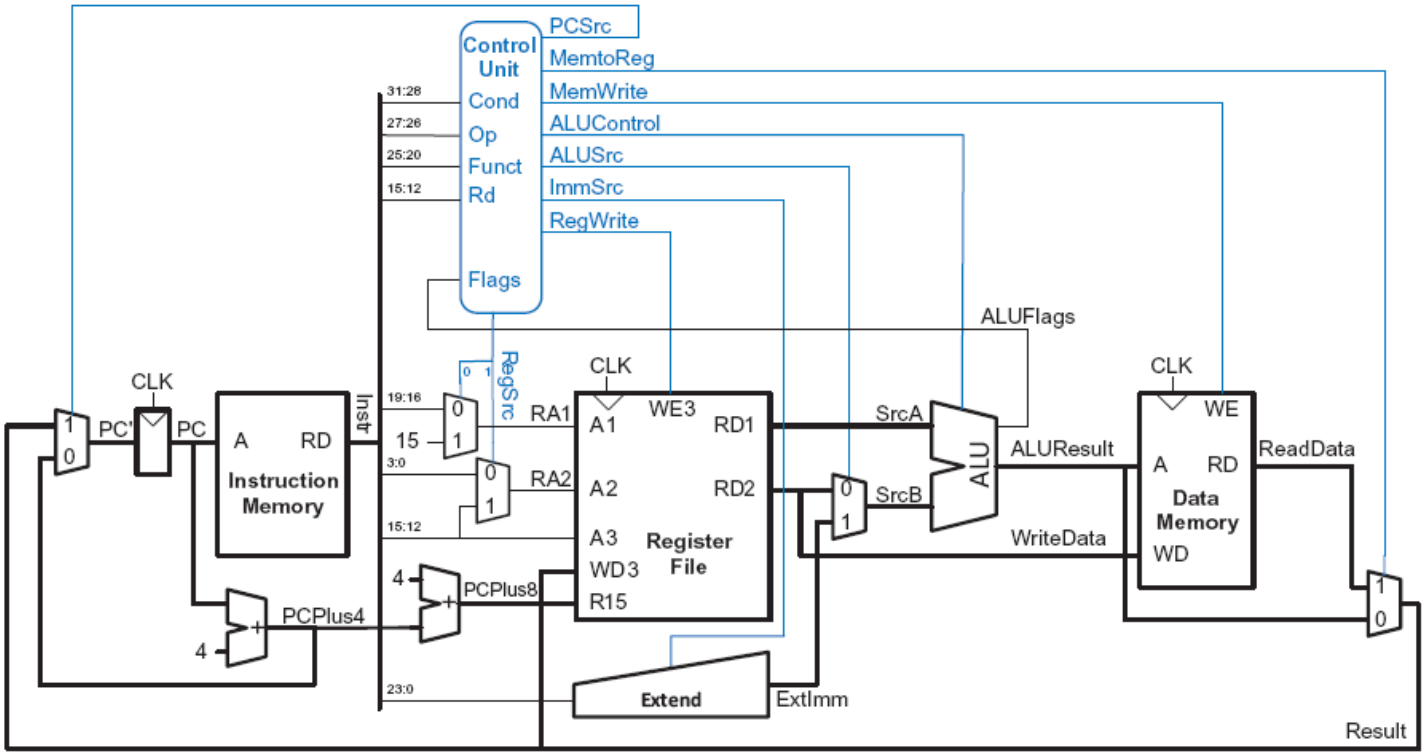


Unità di controllo

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.



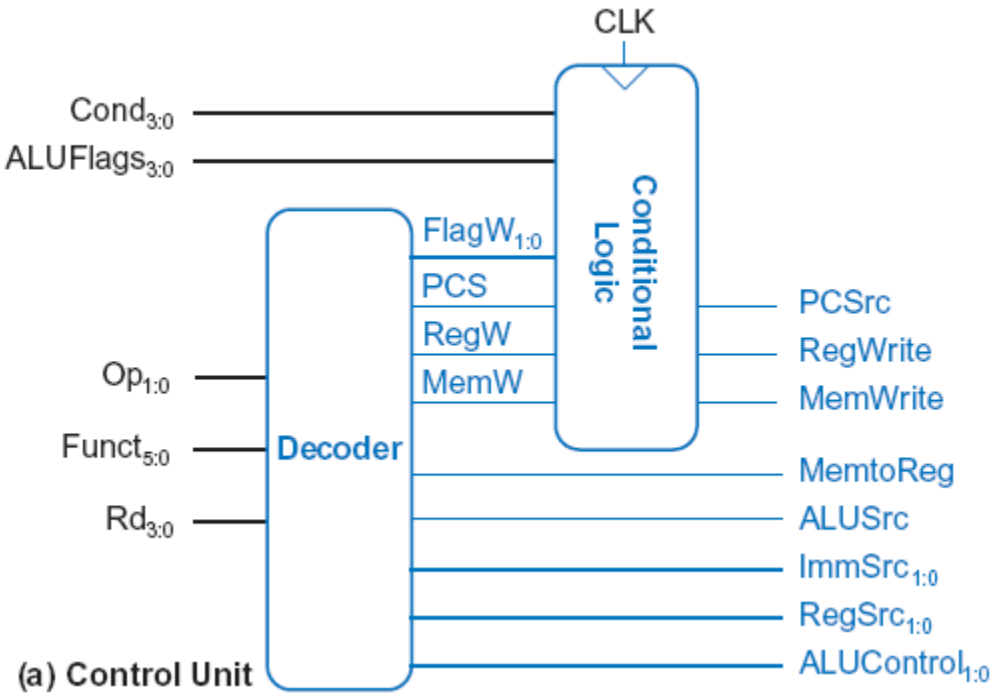
Unità di controllo

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:

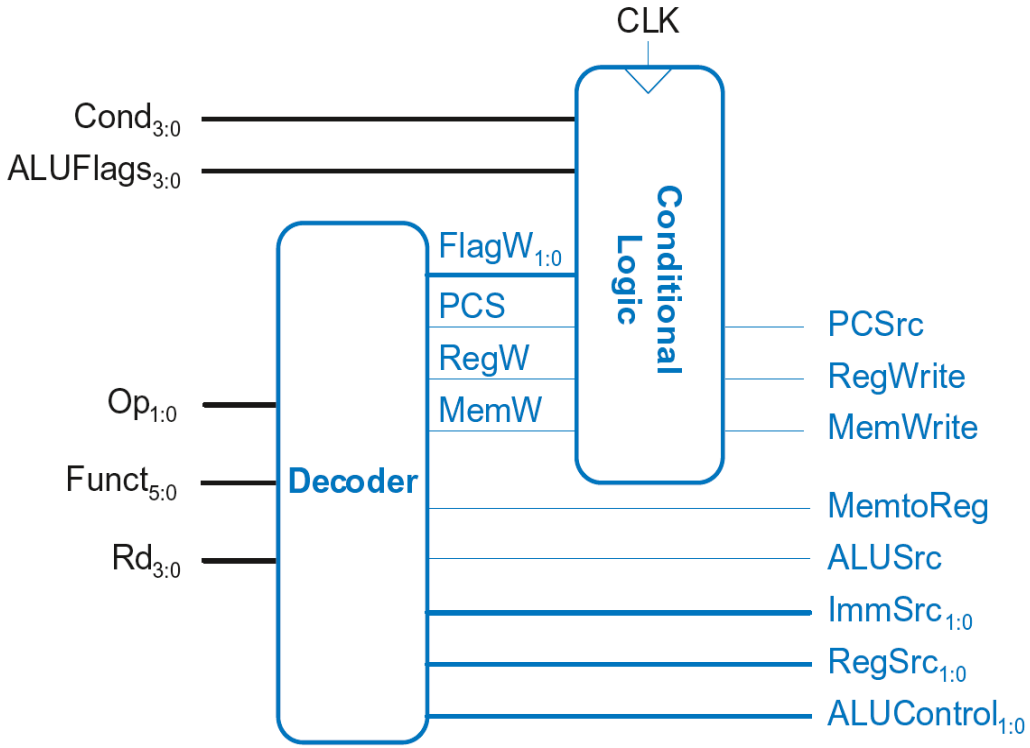
- un **decodificatore principale**, che produce la maggior parte dei segnali di controllo;
- un **decoder ALU**, che utilizza il campo Funct per determinare il tipo di istruzione data-processing;
- la **logica di controllo del PC**, che determina se il PC deve essere aggiornato a causa di una istruzione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



Unità di controllo

Dividiamo l'unità di controllo in due parti principali:

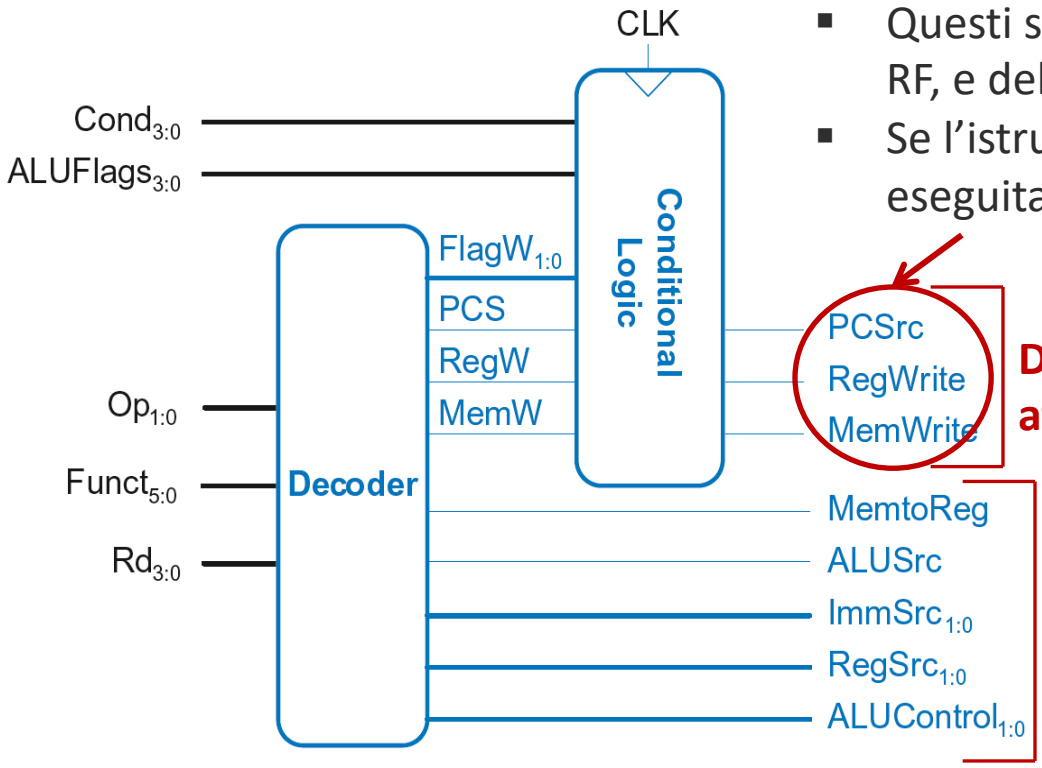
▶️ il **decoder** – genera i segnali di controllo sulla base dei campi di Instr;
▶️ la **logica condizionale** – gestisce i flag di stato e li aggiorna quando l'istruzione deve essere eseguita su condizione.



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.

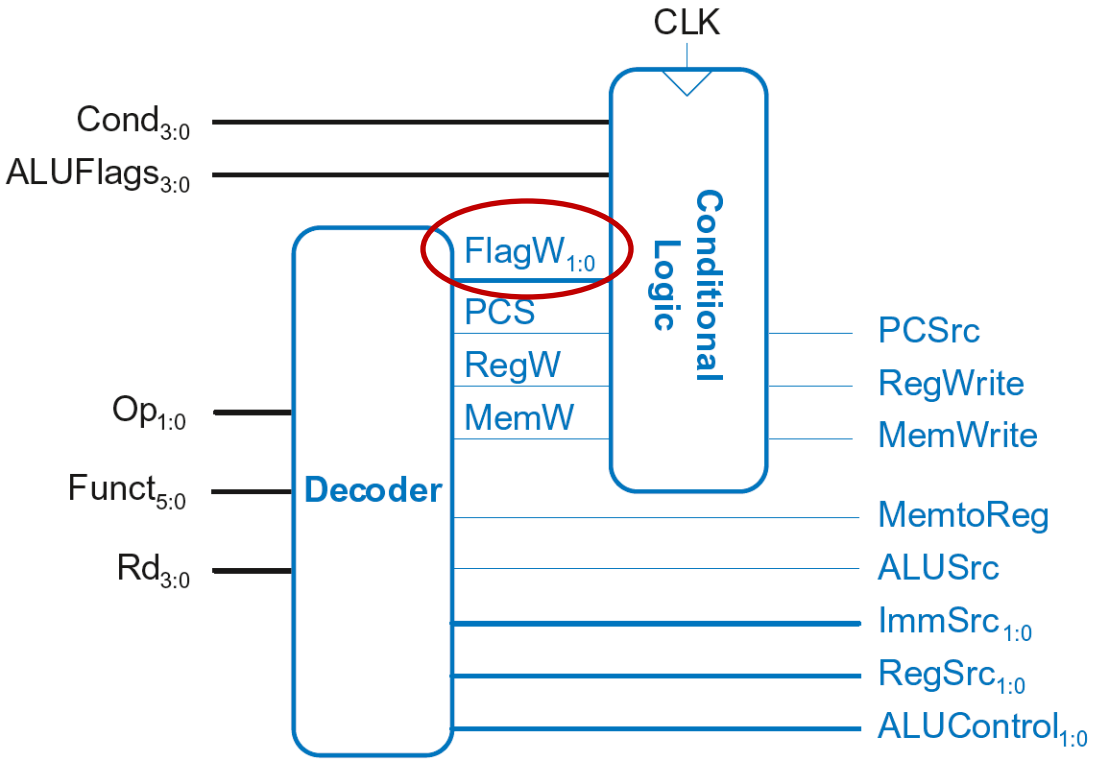


- Questi segnali cambiano lo stato di PC, RF, e della memoria
- Se l'istruzione non deve essere eseguita sono forzati a 0

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**.

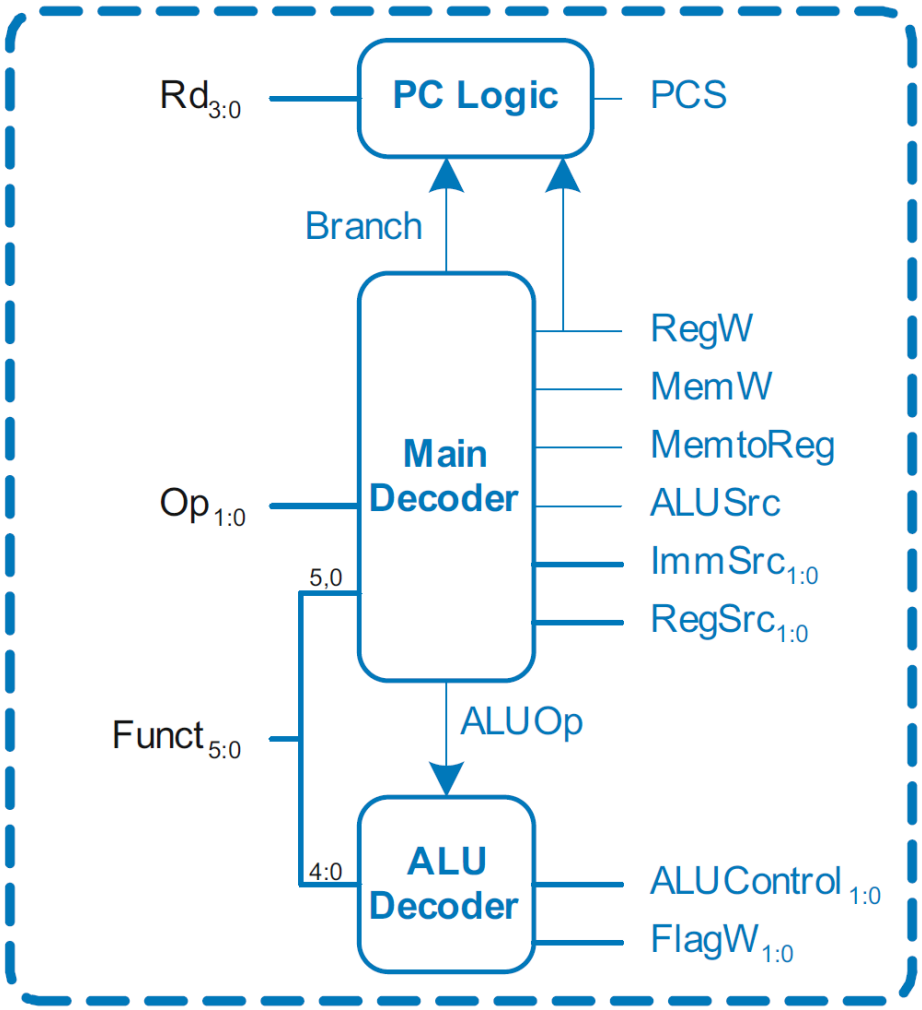
▶️ generare i segnali che abilitano la scrittura (**MemW** e **RegW**), i quali devono passare attraverso la logica condizionale prima di diventare segnali datapath (**MemWrite** e **RegWrite**). Tali segnali possono essere azzerati dalla logica condizionale, se la condizione non è soddisfatta.

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

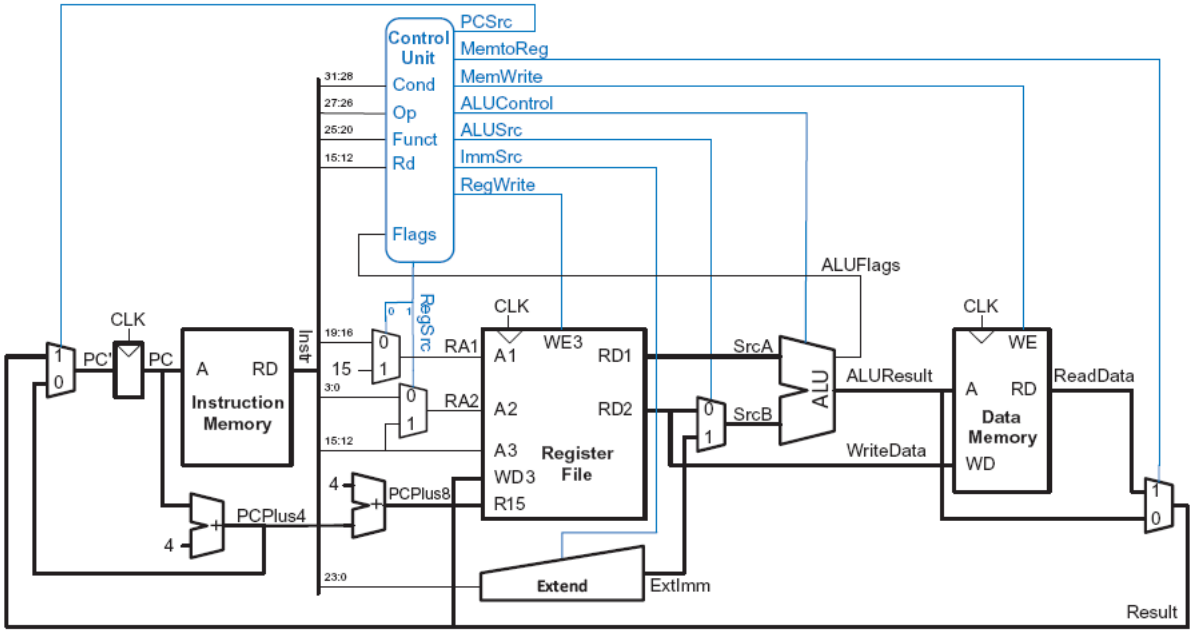
La logica per il decoder principale può essere sviluppata dalla tabella di verità utilizzando le tecniche standard per la progettazione della logica combinatoria.

Single-Cycle Control: Decoder

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

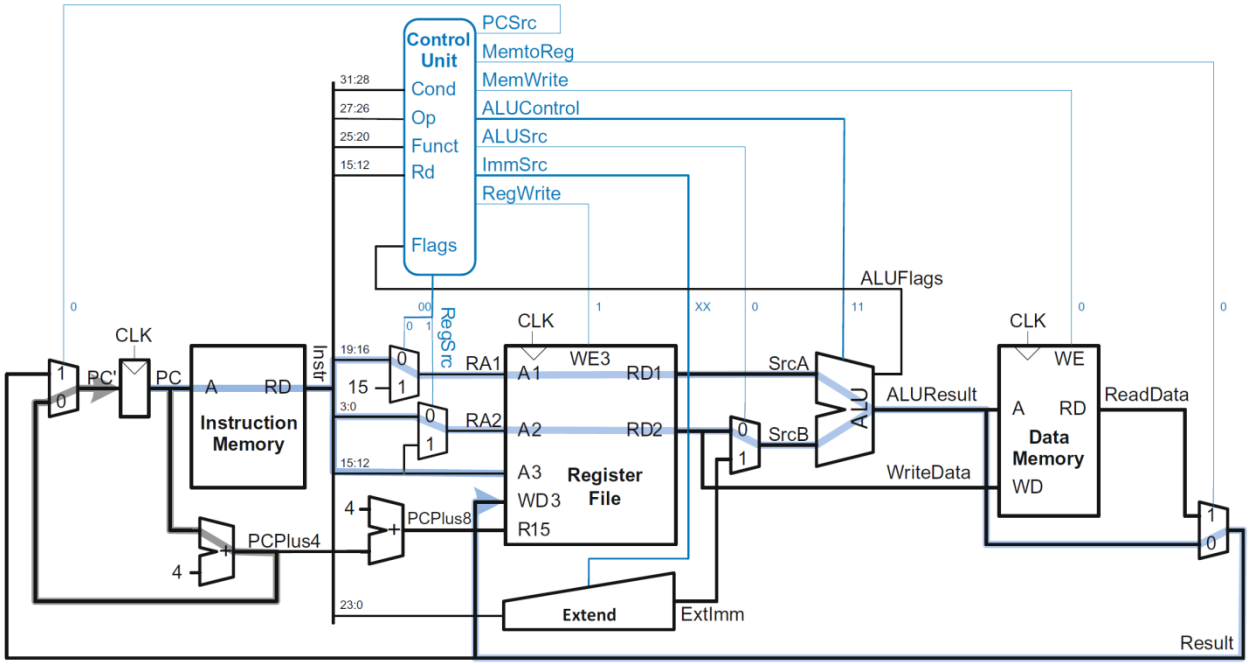


Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
11	X	X	B	1	0	0	1	10	0	X1	0



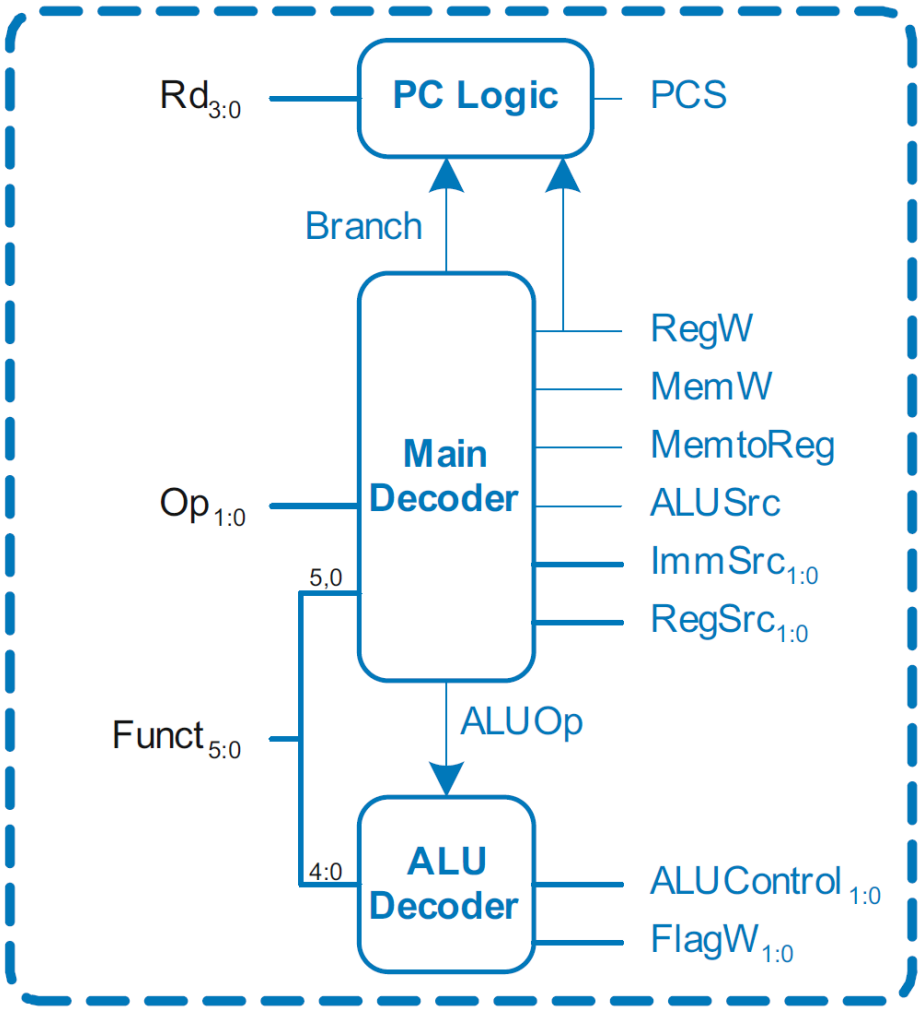
Example: ORR

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



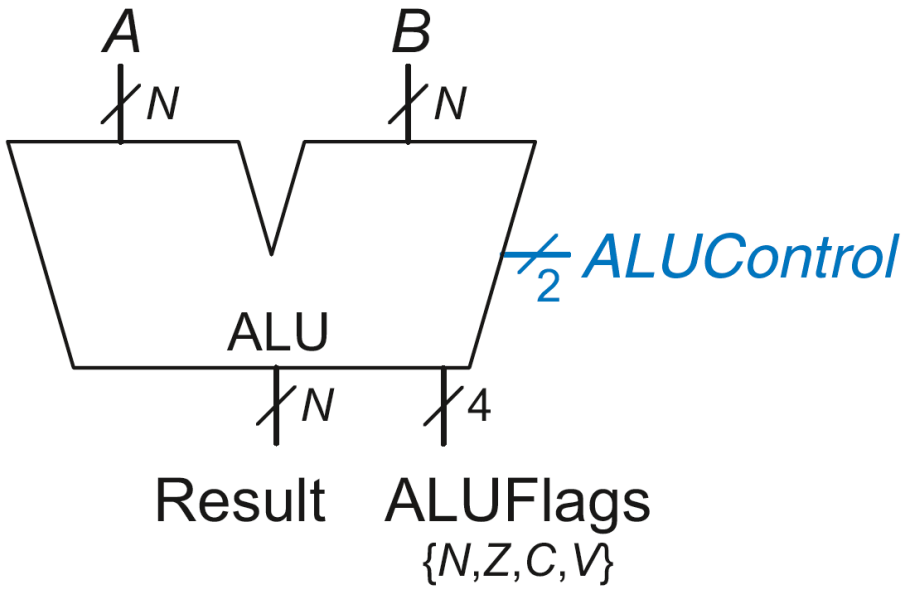
Single-Cycle Control: Decoder

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



Review: ALU

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



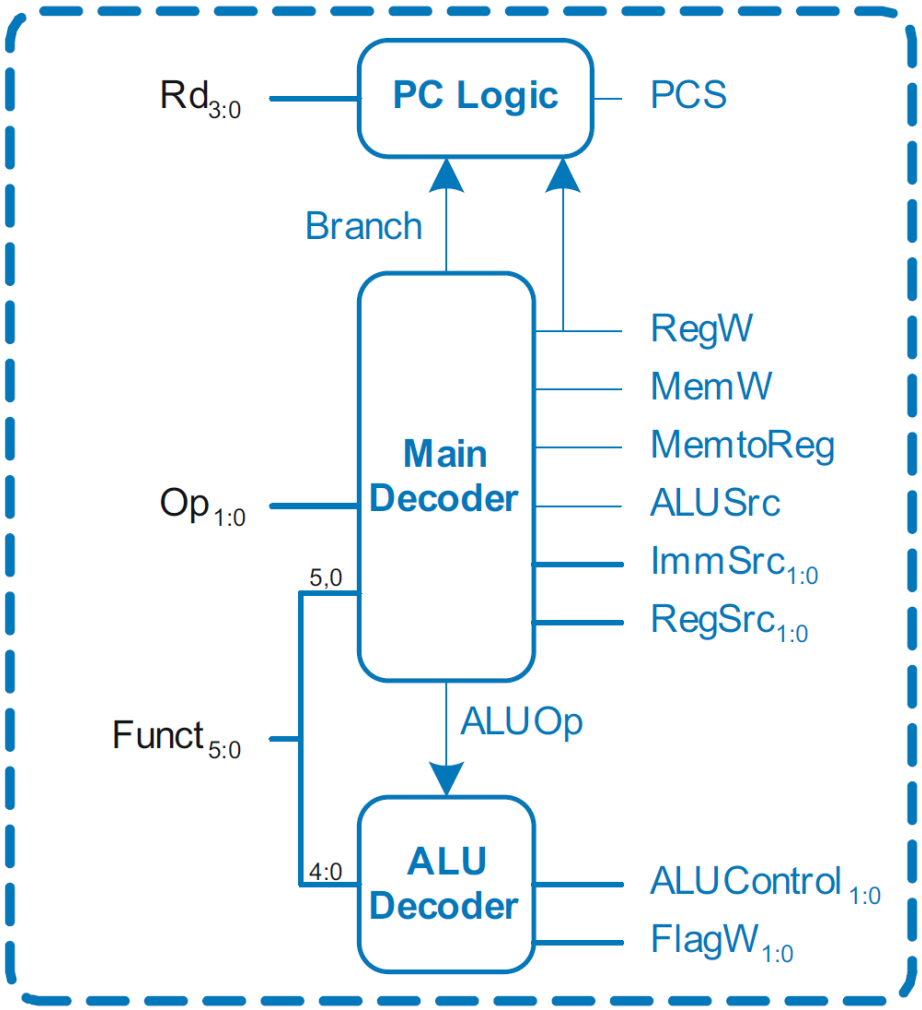
Control Unit: ALU Decoder

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

Single-Cycle Control: Decoder

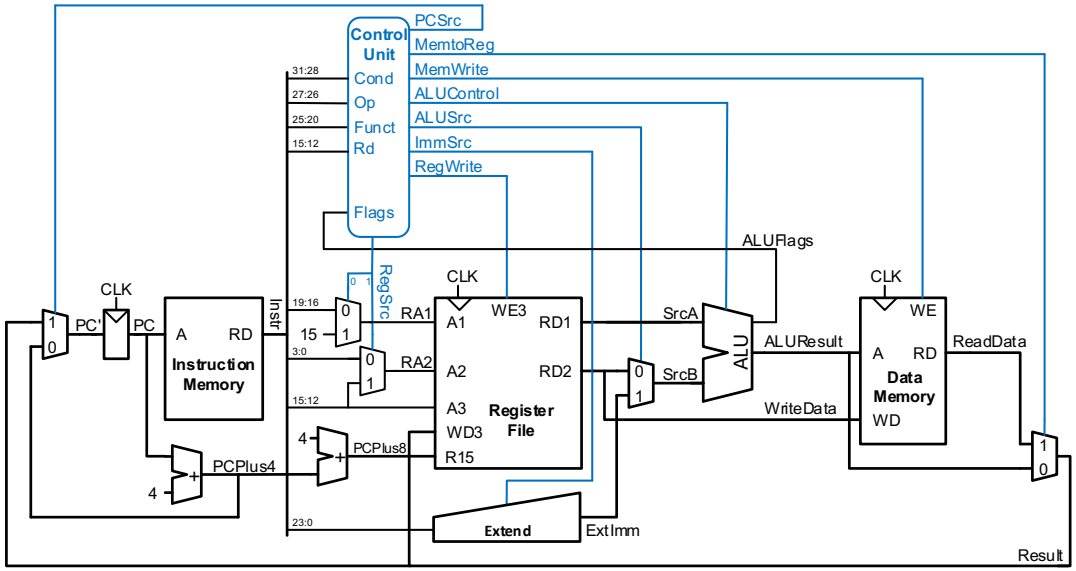
- Main Decoder
- ALU Decoder
- **PC Logic**



Single-Cycle Control: PC Logic

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

PCS = ((*Rd* == 15) AND *RegW*) OR *Branch*

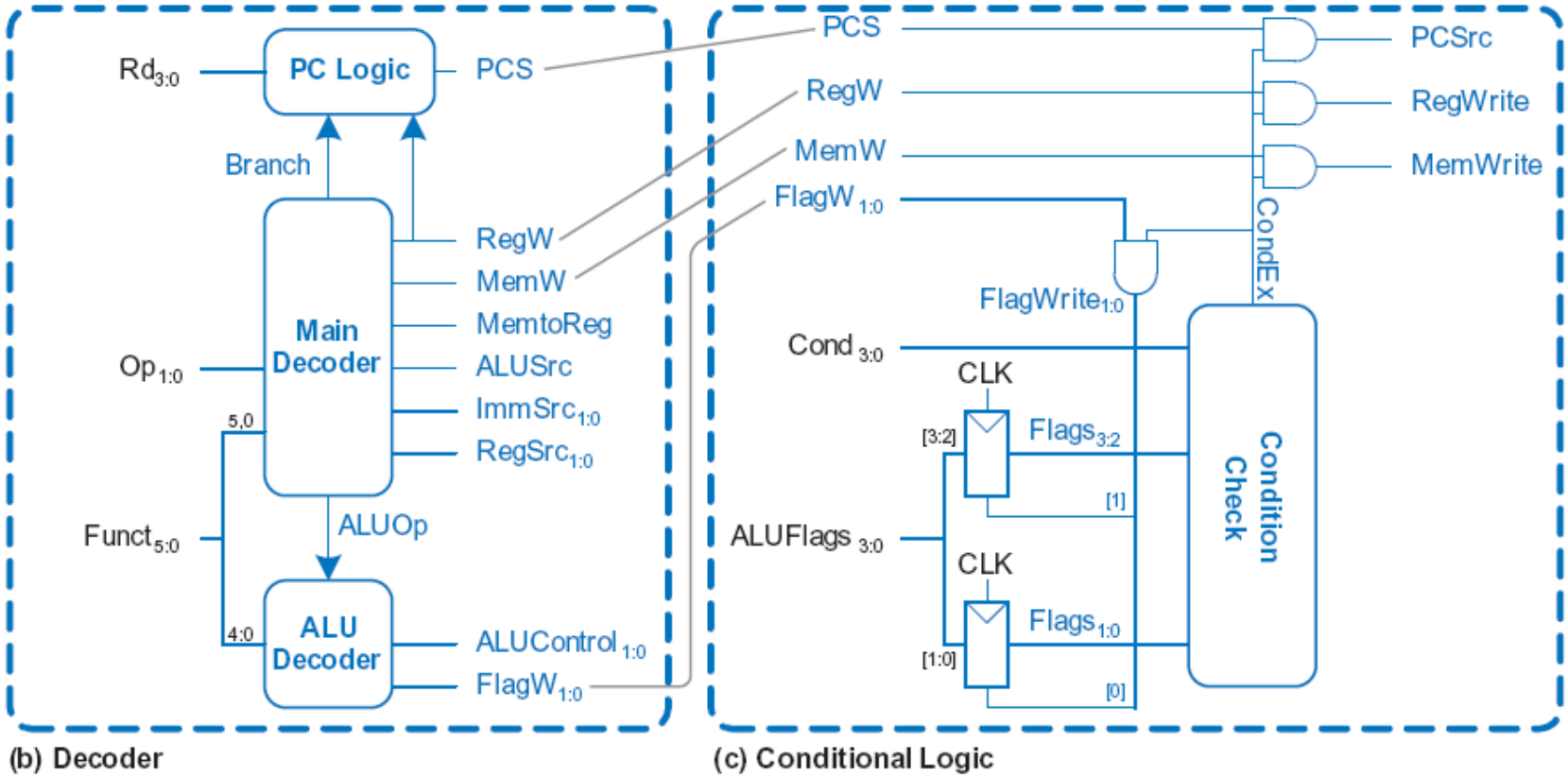


Se l'istruzione è eseguita: ***PCSrc*** **=** ***PCS***

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	$\bar{Z}$

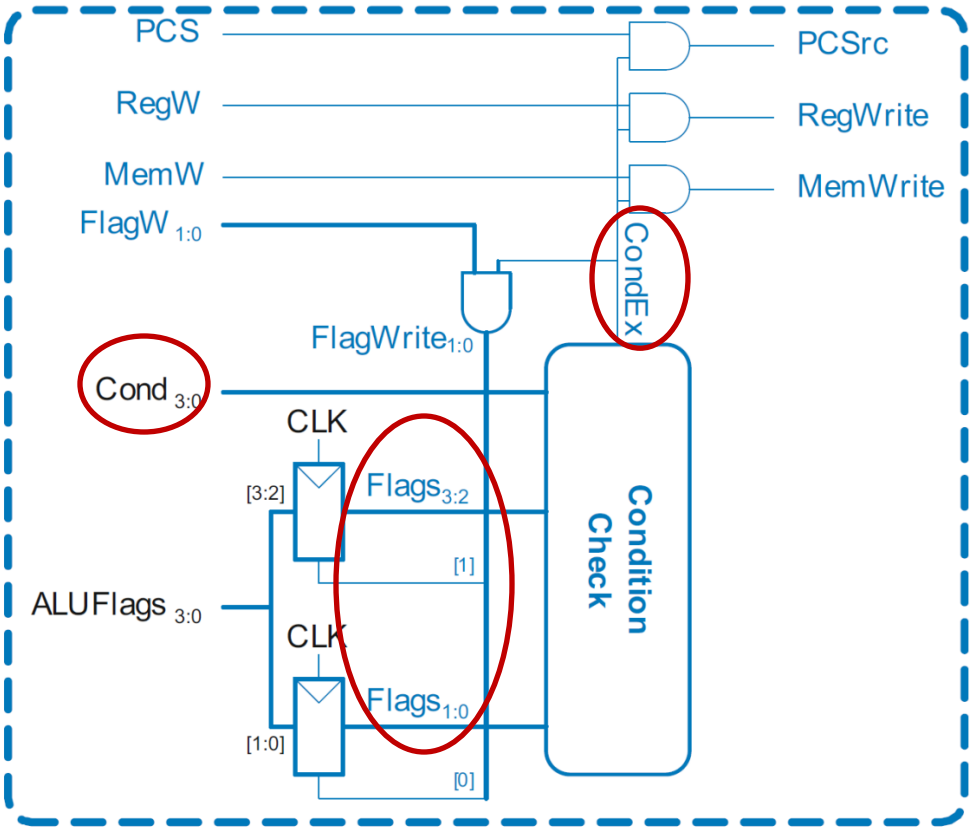
A==B sse A-B==0 sse Z=1

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

Conditional Logic: Conditional Execution

$Flags_{3:0}$ = NZCV



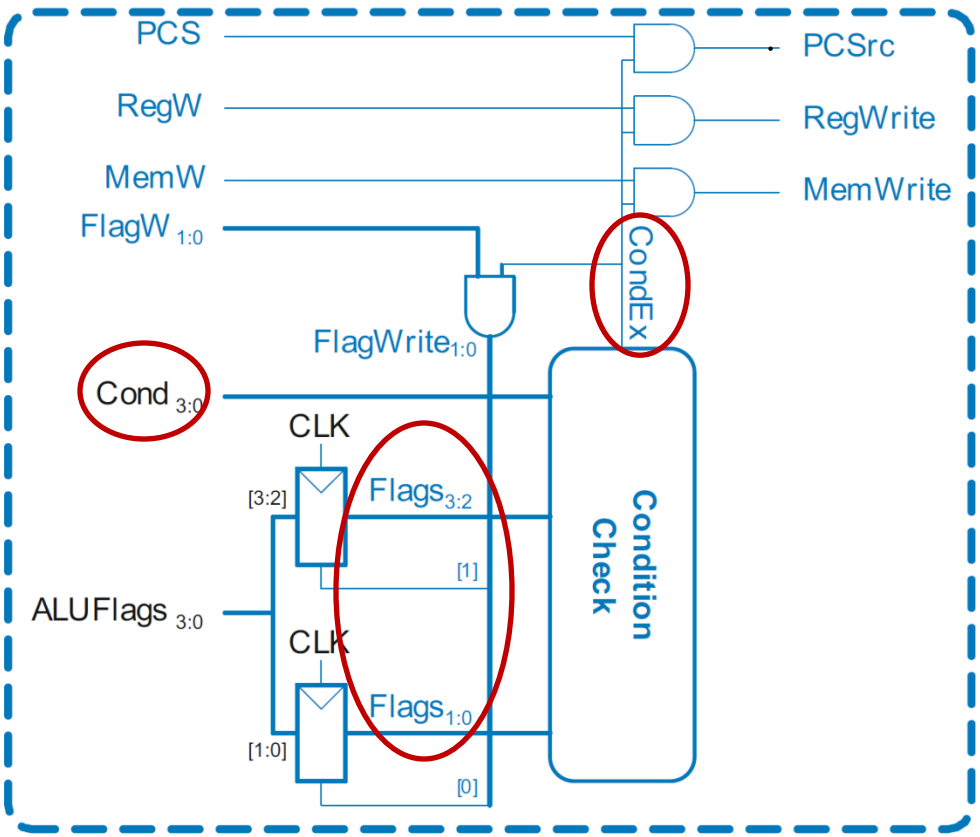

```
AND R1, R2, R3
```

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

CondEx = 1

Conditional Logic: Conditional Execution

Flags$_{3:0}$ = NZCV

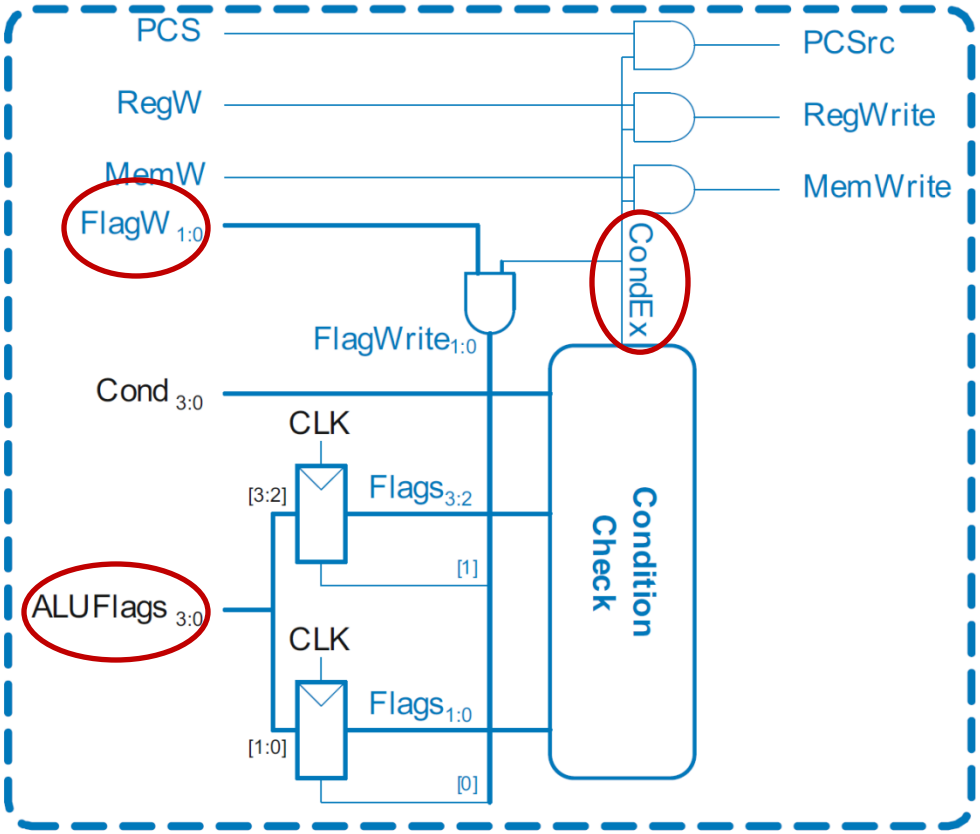



```
EOREQ R5, R6, R7
```

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

Conditional Logic: Update (Set) Flags

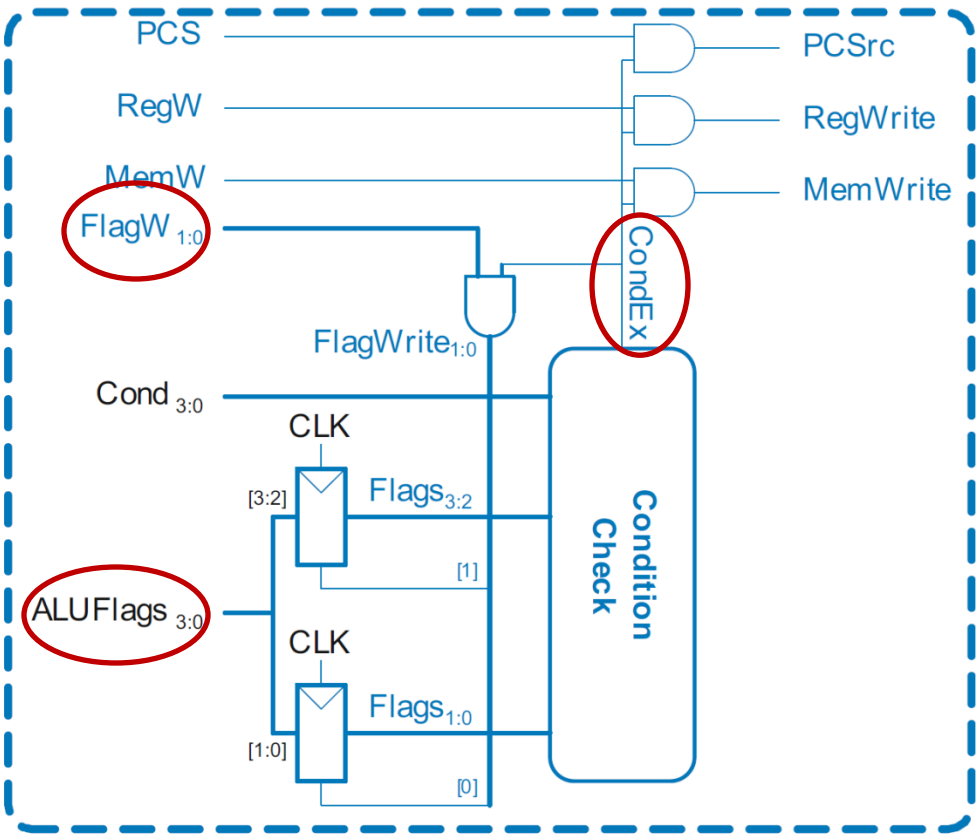
$Flags_{3:0}$ = NZCV



$Flags_{3:0}$ vengono aggiornati con i valori di $ALUFlags_{3:0}$ se si verificano le seguenti condizioni:

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

Conditional Logic: Update (Set) Flags

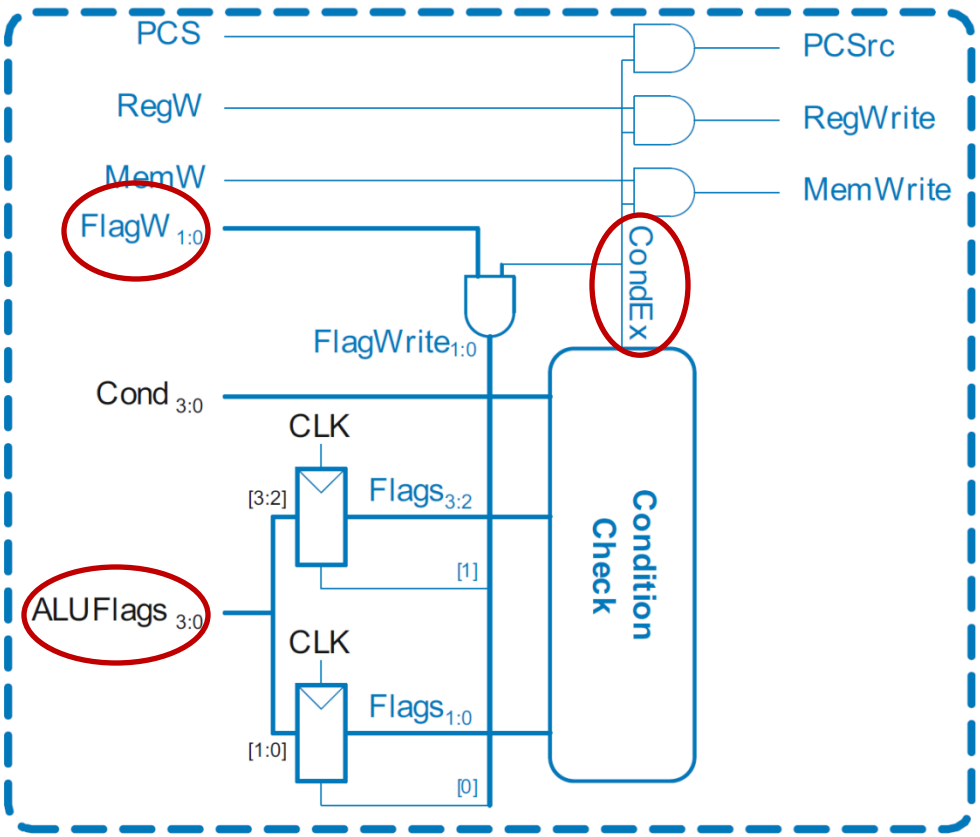



```
SUBS R5, R6, R7
```

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

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

Conditional Logic: Update (Set) Flags

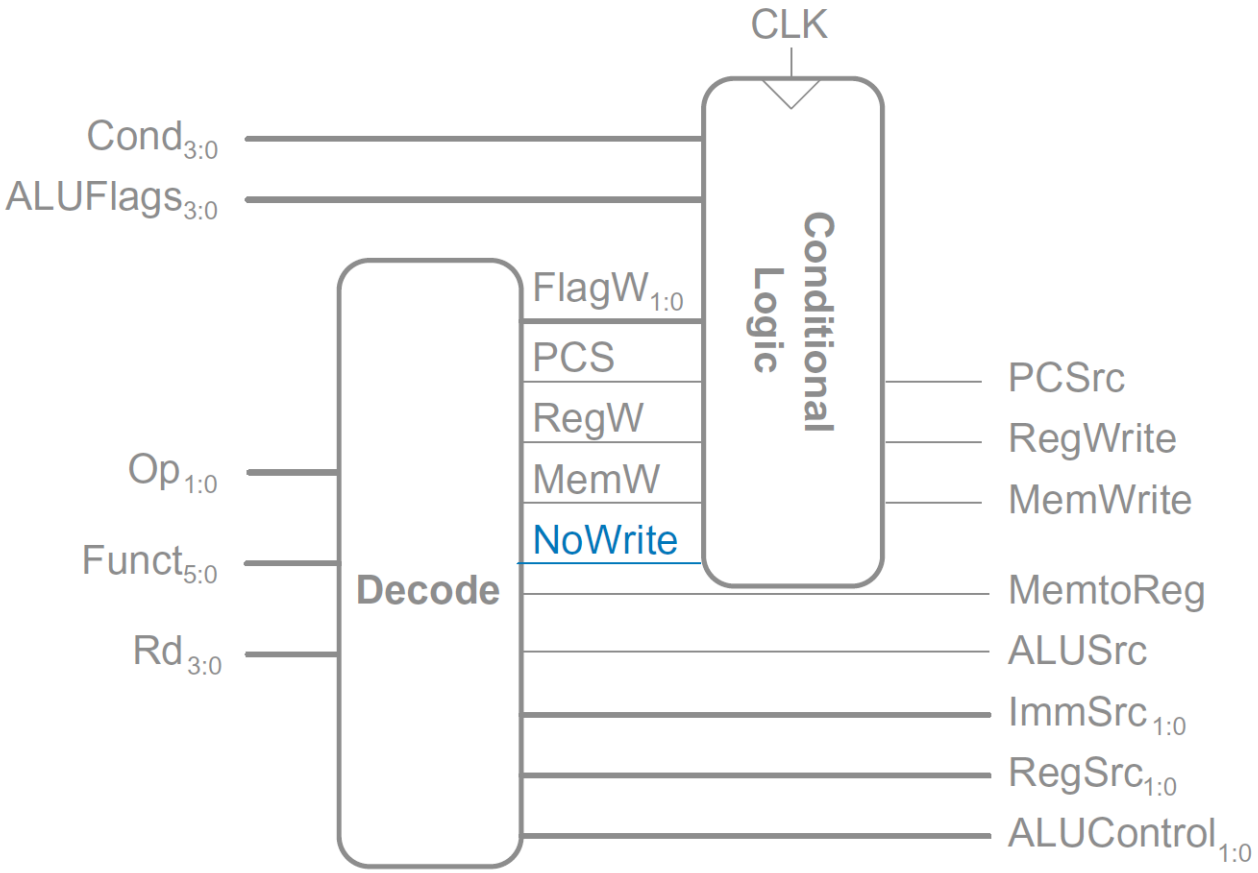



```
ANDS R7, R1, R3
```

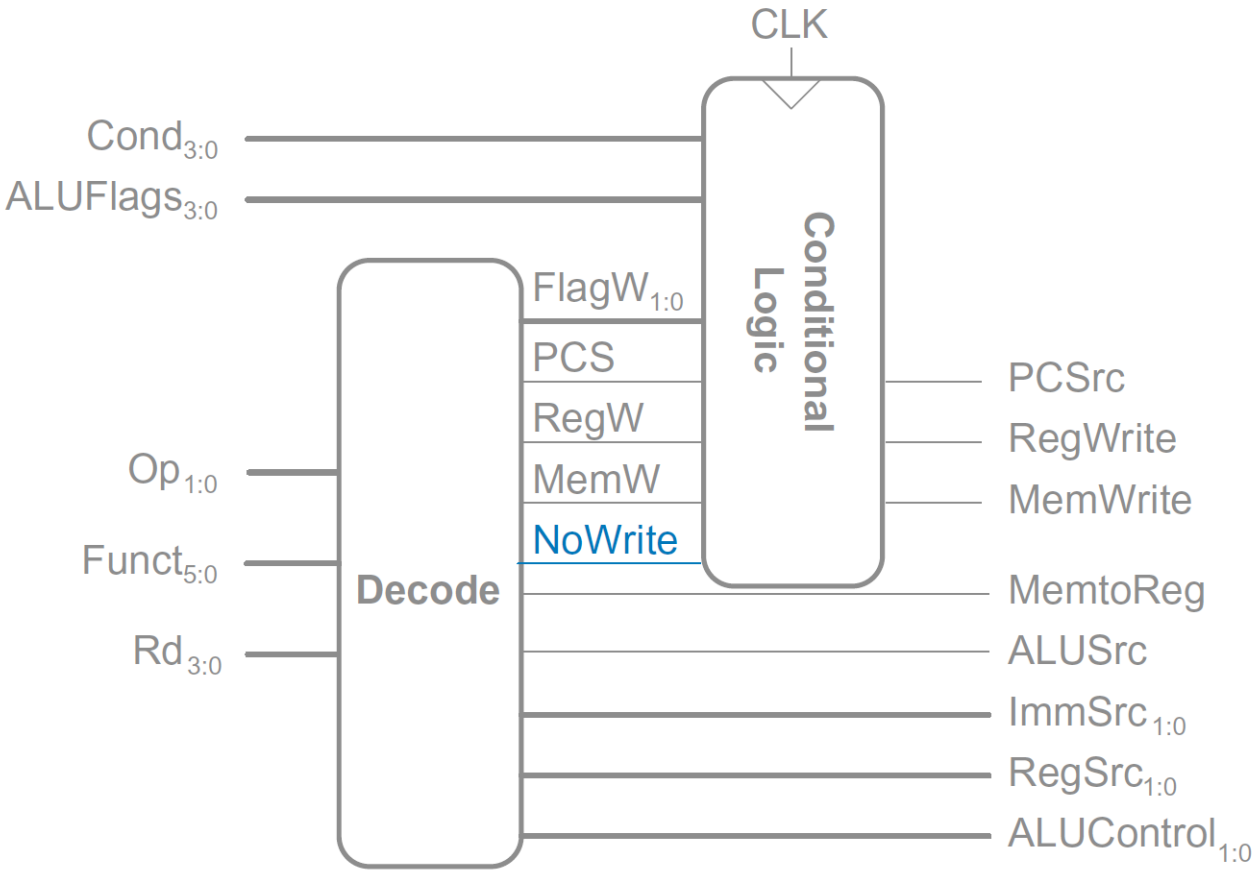
FlagW$_{1:0}$ = 10 E **CondEx** = 1 (istruzione incondizionata) =>

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

Extended Functionality: CMP

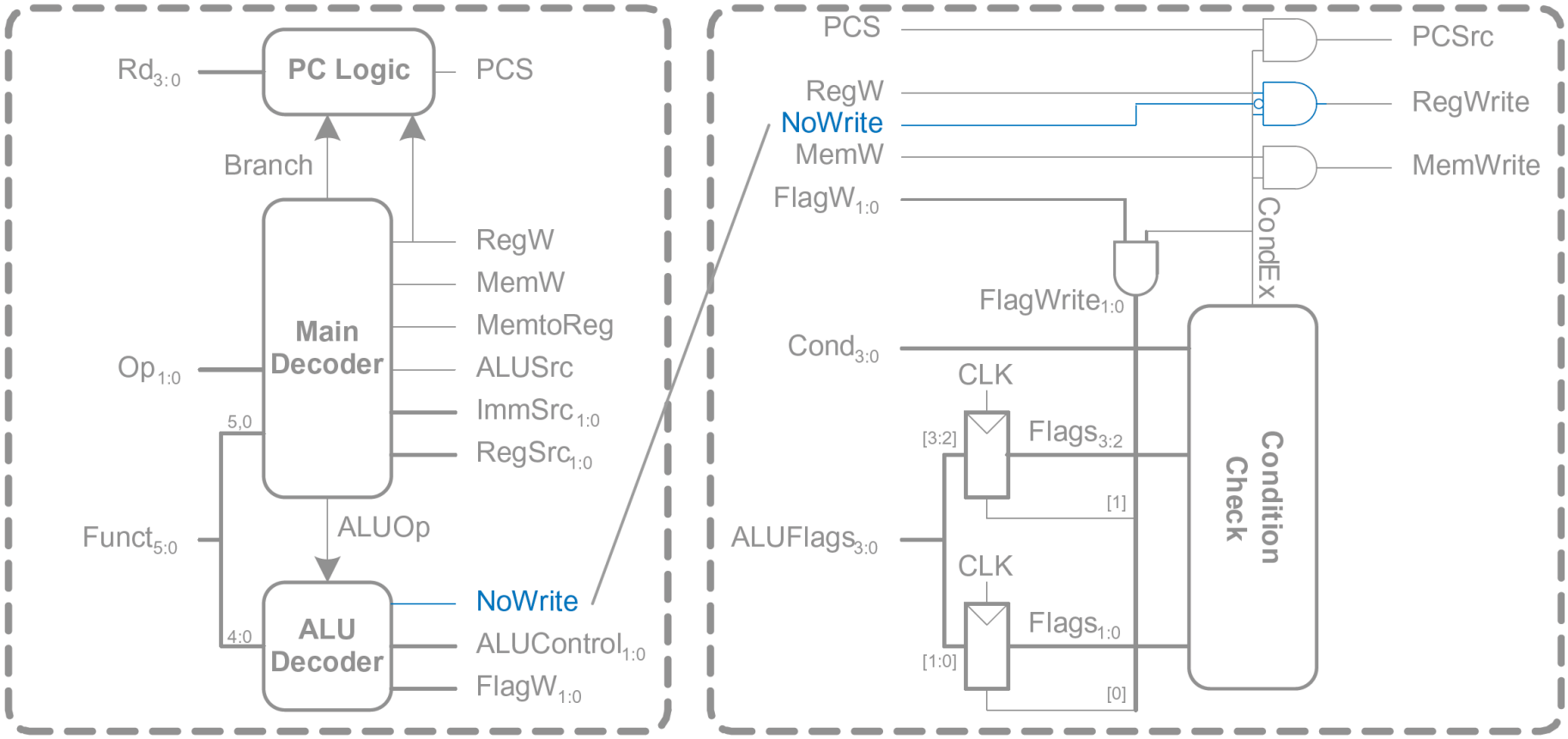


Extended Functionality: CMP



No change to datapath

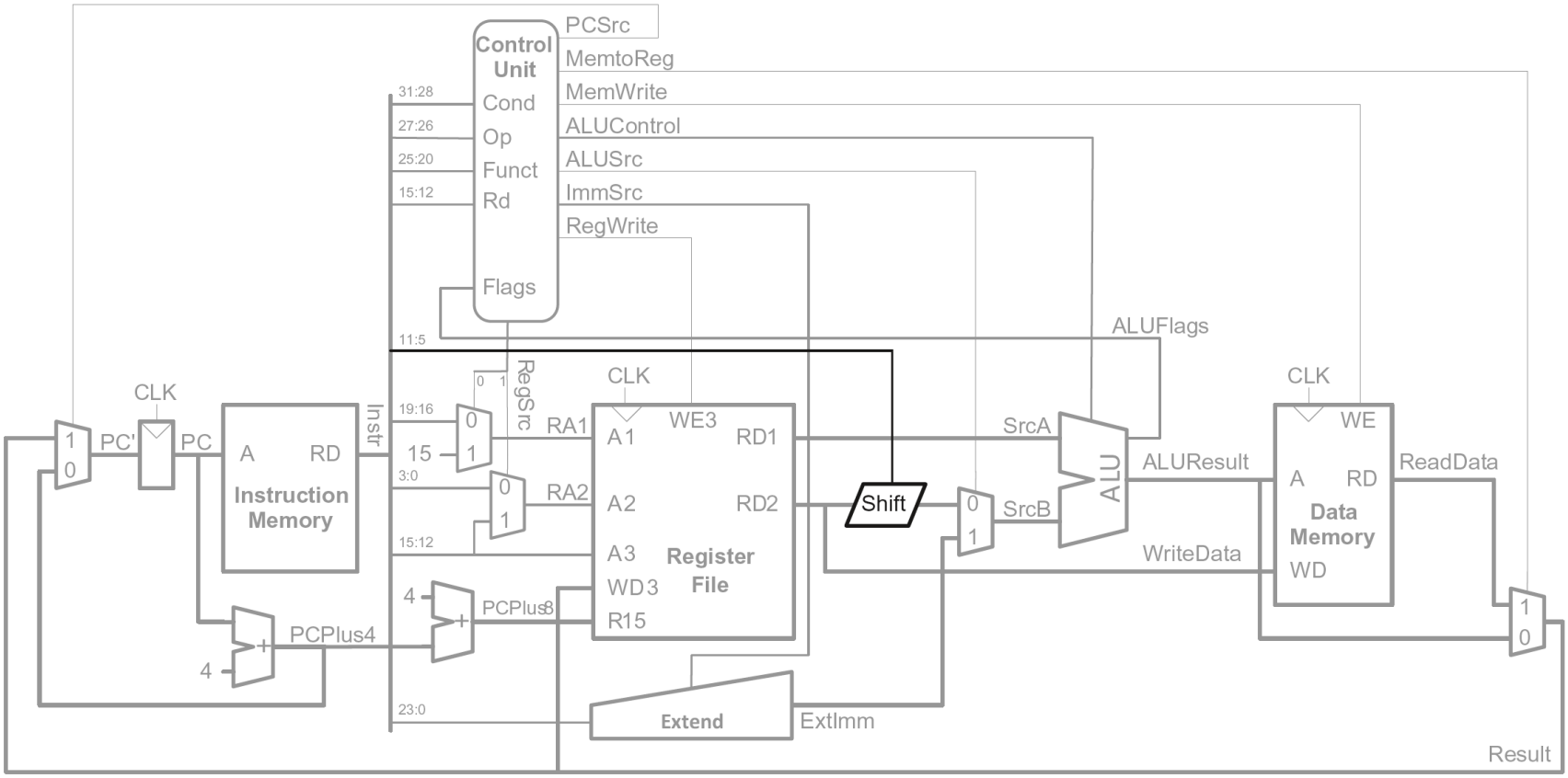
Extended Functionality: CMP



Extended Functionality: CMP

ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$	NoWrite
0	X	X	Not DP	00	00	**0**
1	0100	0	ADD	00	00	**0**
		1			11	**0**
	0010	0	SUB	01	00	**0**
		1			11	**0**
	0000	0	AND	10	00	**0**
		1			10	**0**
	1100	0	ORR	11	00	**0**
		1			10	**0**
	1010	**1**	**CMP**	**01**	**11**	**1**

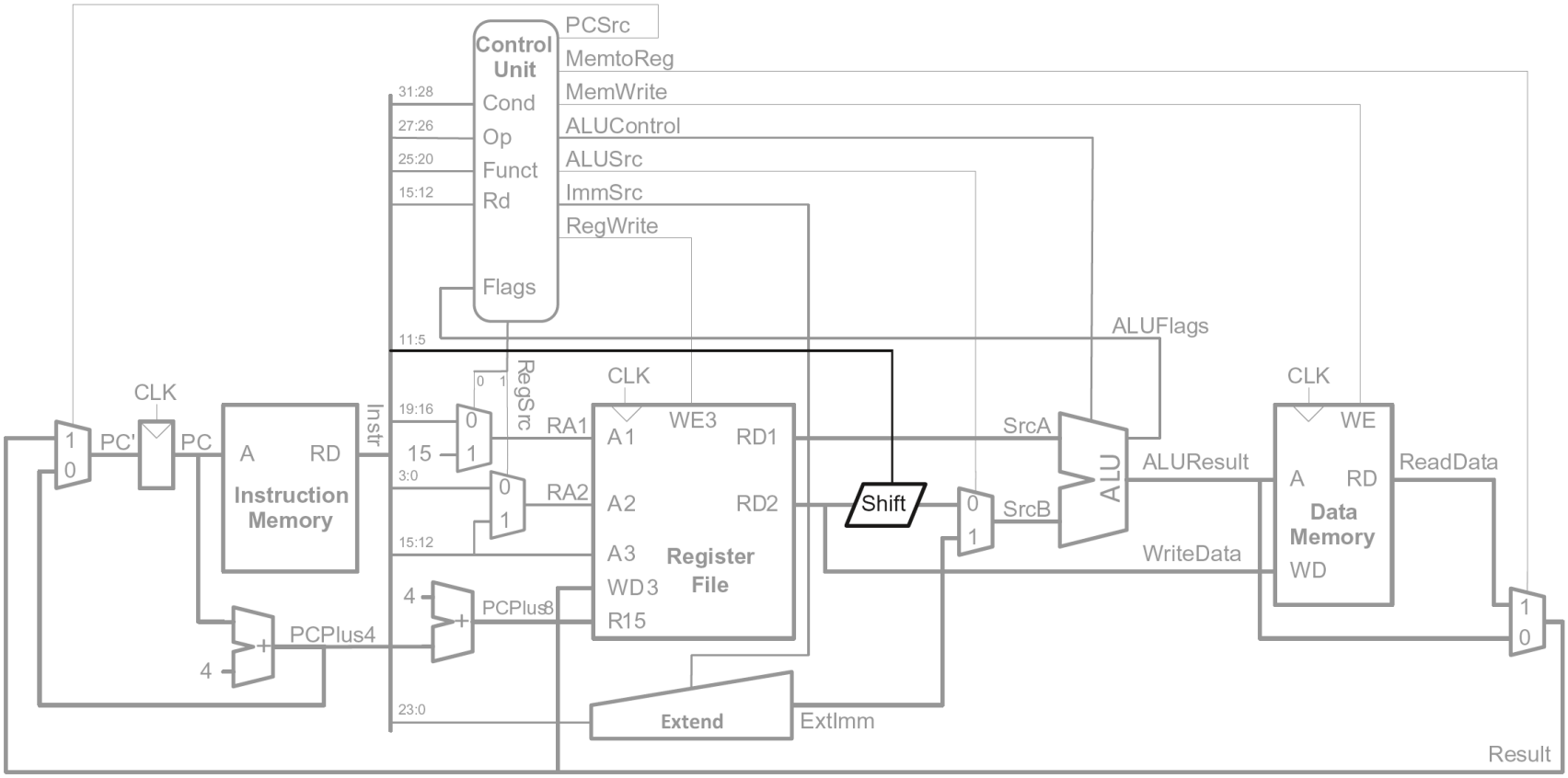
Extended Functionality: Shifted Register



ADD R7, R2, R12, LSR #5

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
14	0	0	4	0	2	7	5	01_2	0	12
cond	op	I	cmd	S	rn	rd	shamt5	sh		rm

Extended Functionality: Shifted Register



No change to controller

Review: Processor Performance

Program Execution Time

= (#instructions)(cycles/instruction)(seconds/cycle)

= # instructions x CPI x T_C

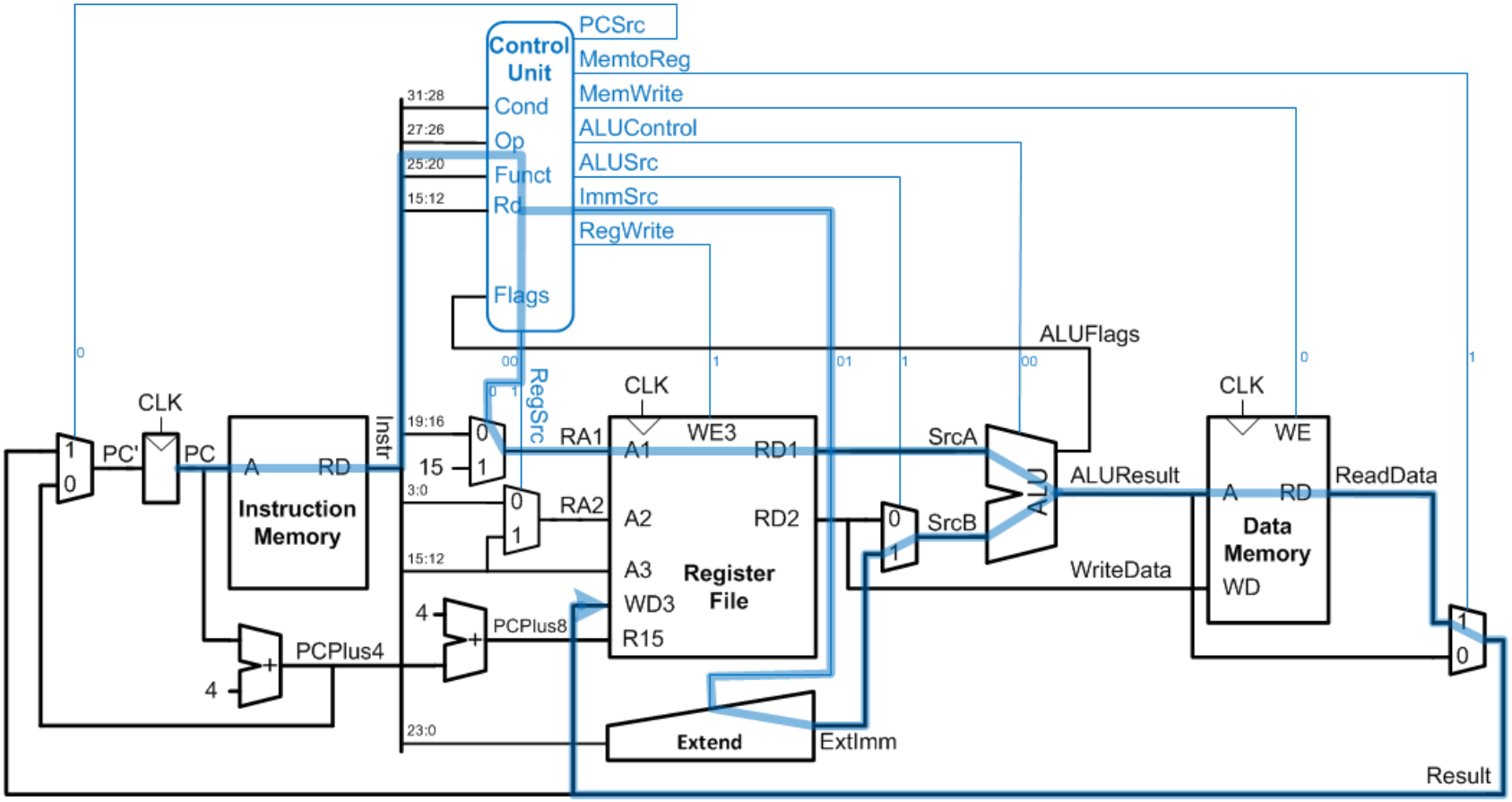
Analisi delle prestazioni

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.

Single-Cycle Performance



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

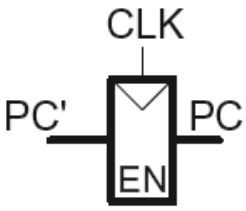
Il **datapath** è sviluppato in modo incrementale, i nuovi collegamenti sono evidenziati in **nero** (o **blu**), mentre quanto già introdotto è rappresentato in **grigio**.

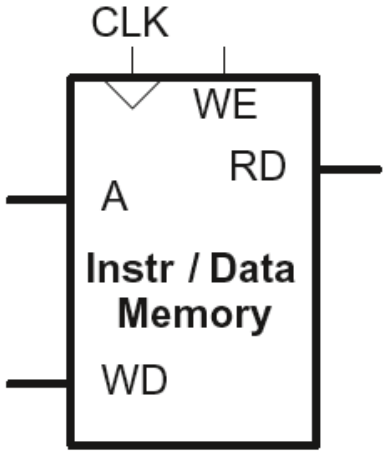
Al fine di facilitare la comprensione dell'architettura di un processore ARM, considereremo un limitato set di istruzioni:

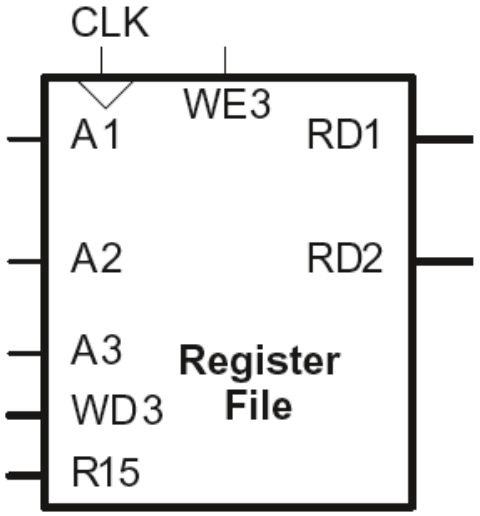
- ▶️ le istruzioni di elaborazione dati: **ADD**, **SUB**, **AND**, **ORR** (con registro e modalità di indirizzamento diretto e senza shift);
- ▶️ le istruzioni di Memoria: **LDR**, **STR** (diretto e con offset positivo);
- ▶️ le istruzioni di salto (branch): **B**.

Multicycle State Elements

Replace Instruction and Data memories with a single unified memory – more realistic



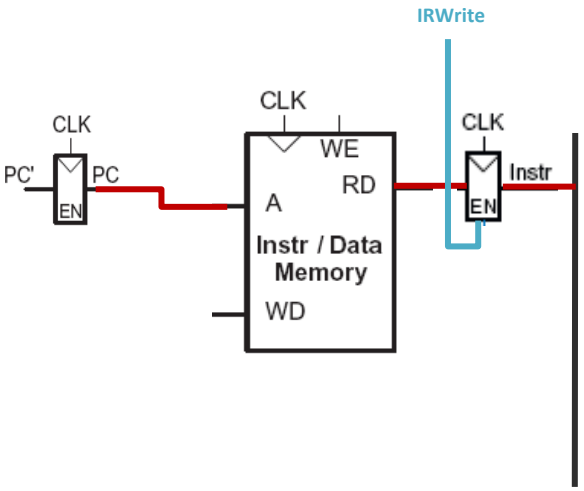


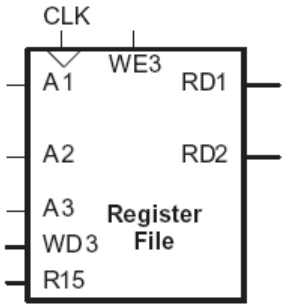


Datapath LDR

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.



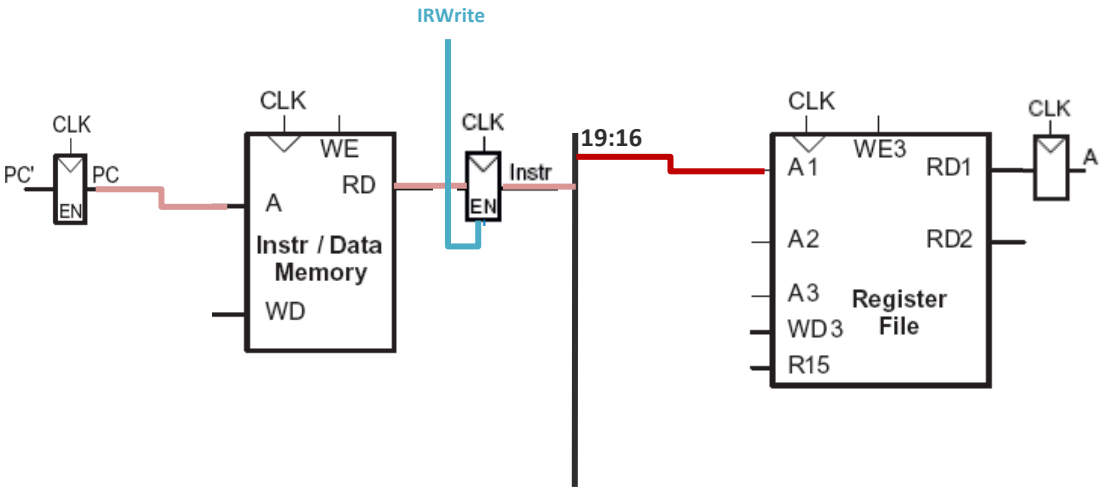


Datapath LDR

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**.

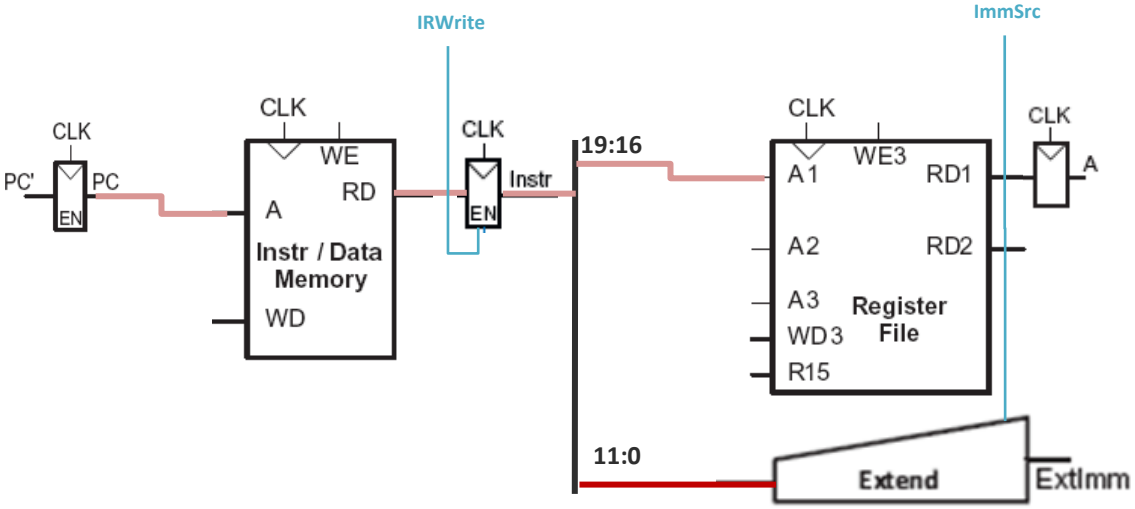


Datapath LDR

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.

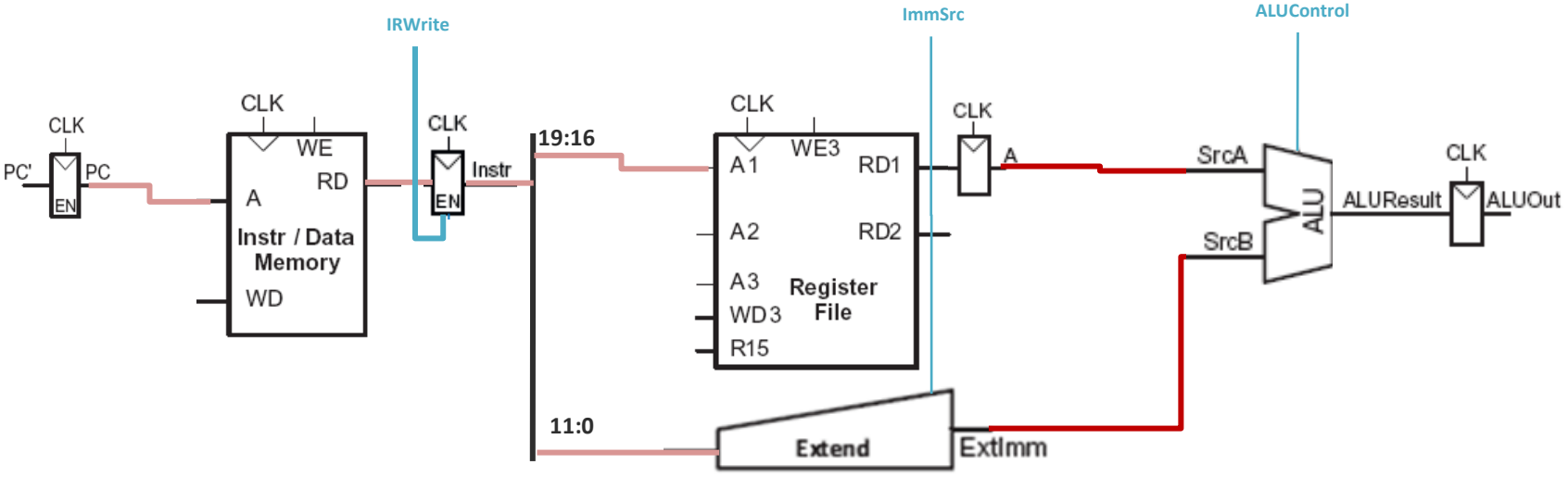


Datapath LDR

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).

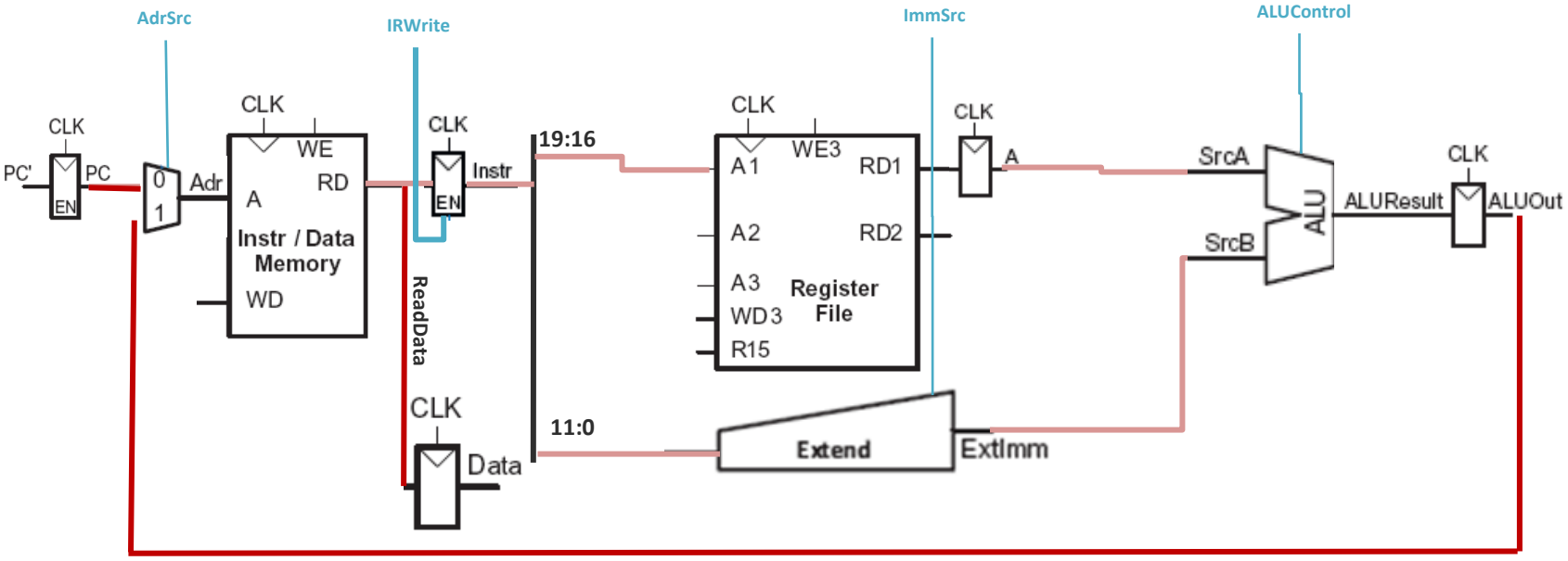
La **ALU** genera un valore a 32 bit **ALUResult**, che viene memorizzato in un registro **ALUOut**.



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**.

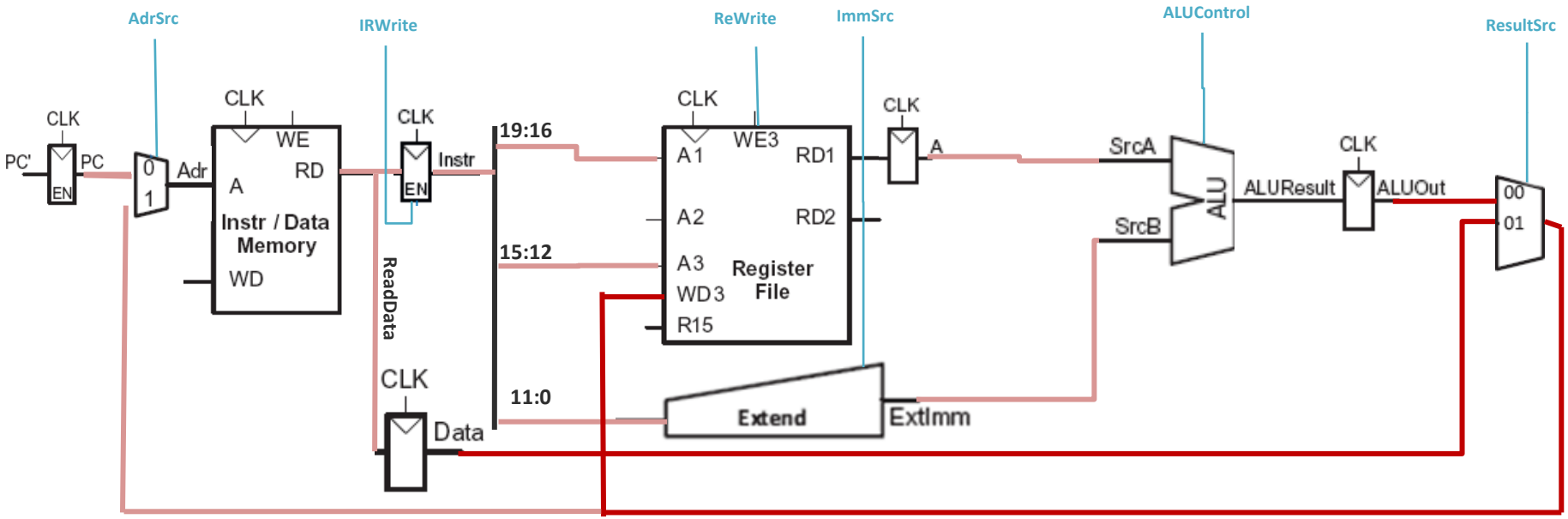


Datapath LDR

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**.

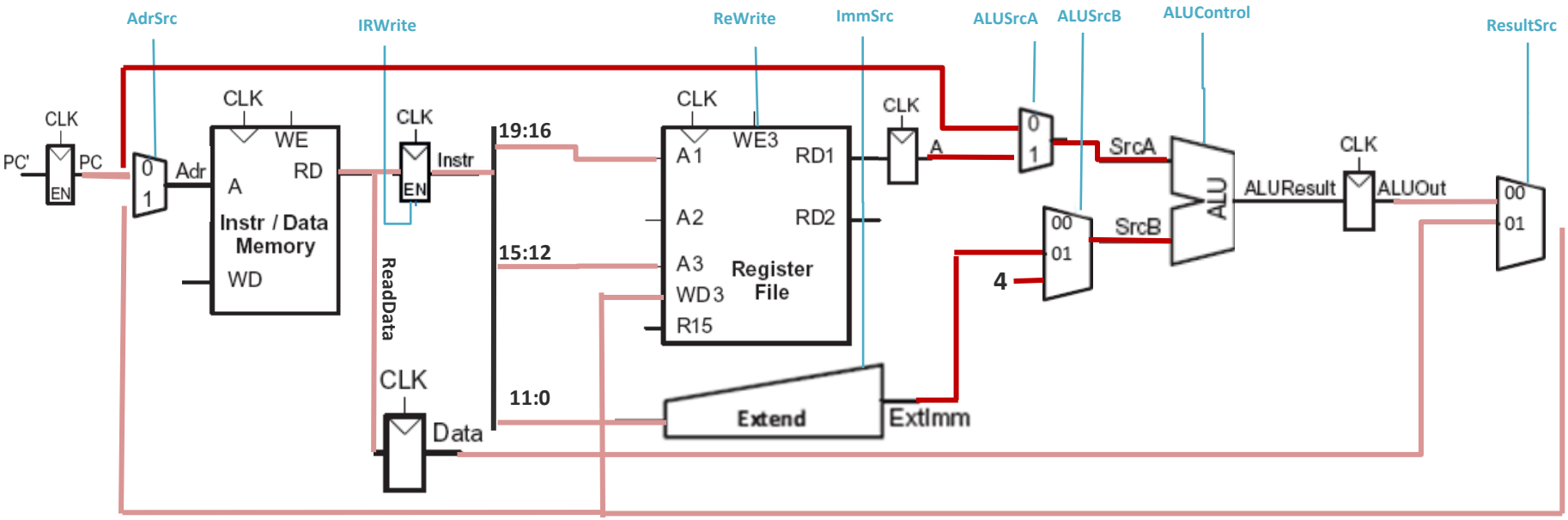
Il segnale **RegWrite** deve essere impostato a **1**, per permettere la scrittura nel registro.

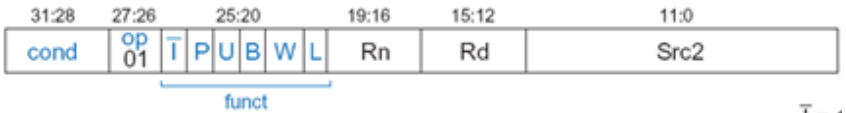


Datapath LDR

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.