

5. MICROARCHITETTURA

L'architettura rappresenta la struttura di un calcolatore dal punto di vista di un programmatore. È definita dal set di istruzioni e dagli operandi che vengono codificati in linguaggio macchina. Al linguaggio macchina corrisponde un linguaggio assembly ossia un formato human-readable delle istruzioni, rispetto a quello macchina che è un formato computer-readable (1 e 0). Esistono molte architetture differenti tra cui ARM e X86

La **MICROARCHITETTURA** rappresenta la congiunzione fra i circuiti logici e l'architettura: **definisce la disposizione specifica di registri, ALU, macchine a stati finiti, le memorie e altri blocchi logici necessari per implementare una architettura**

Due microarchitetture possono realizzare la medesima architettura ma con soluzioni tecnologiche e prestazioni molto differenti. Ad esempio Intel e AMD realizzano diversi modelli (microarchitetture) della medesima architettura x86.

Anche l'architettura ARM può avere diverse microarchitetture, caratterizzate da diversi rapporti prestazioni/costo/complessità. Tutte devono essere in grado di eseguire gli stessi programmi, ma la loro struttura interna può essere anche molto diversa.

L'architettura ARM è stata sviluppata negli anni 80 dalla Advanced RISC Machines ed è impiegata soprattutto nel settore degli smartphone e tablet.

Principi di design dell'architettura ARM

L'architettura ARM si basa su 4 principi di progettazione fondamentali RMSG:

1. Regularity supports design simplicity (la regolarità favorisce la semplicità)
2. Make the common case fast (rendere veloci le cose frequenti)
3. Smaller is faster (più piccolo è più veloce)
4. Good design demands good compromises (un buon progetto richiede buoni compromessi).

Applicazione del 1° principio ARM: Regularity supports design simplicity

Il **formato costante delle istruzioni arm** è un'applicazione del primo principio Regularity supports design simplicity.

Ogni **istruzione ARM** è rappresentata 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).

In ogni istruzione inoltre è presente un **numero costante di operandi: di solito due operandi sorgenti e un operando destinazione**.

Questo rende le cose più facili da gestire in hardware, perché **i formati delle istruzioni sono consistenti, e l'encoding in hardware è dunque facilitato**.

Applicazione del 2° principio ARM: Make the common case fast

L'uso di **più istruzioni semplici assembly per eseguire attività complesse** è un esempio di applicazione del secondo principio: rendere veloci le cose frequenti.

Infatti l'architettura **ARM** rende veloci le cose frequenti perché **comprende solo istruzioni semplici e di uso frequente**. Il numero di istruzioni diverse è tenuto basso in modo tale che il circuito **hardware richiesto per decodificare ed eseguire le istruzioni sia semplice e veloce**.

Elaborazioni più complesse eseguite più raramente sono realizzate con **sequenze di molteplici istruzioni ARM semplici**. Dunque istruzioni di alto livello più complesse vengono tradotte in più sequenze di semplici istruzioni arm (*ad es. se ho una somma e una sottrazione in sequenza effettuo prima la somma e poi sul risultato ottenuto effettuo la sottrazione ottenendo il risultato finale*).

ARM quindi si basa su un'architettura RISC che ha un insieme limitato di istruzioni semplici. Architetture con molte istruzioni complesse si basano su un architettura CISC (complex instruction set computer) esempio x86. CISC comporta hardware aggiuntivo e sovraccarichi che rallentano anche le istruzioni semplici. **L'architettura RISC minimizza la complessità hardware e la codifica necessaria per le istruzioni mantenendo piccolo l'insieme di diverse istruzioni.**

Applicazione del 3° principio ARM: Smaller is faster

I dati nell' architettura ARM possono essere memorizzati:

- Direttamente all'interno di una istruzione:** si definiscono costanti (**immediates**)
- In registri:** ARM ha solo **16 registri, ognuno costituito da 32 bit**, che sono più veloci della memoria
- In memoria:** Più lenta dei registri ma più capiente, 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.

Infatti le istruzioni hanno bisogno di raggiungere rapidamente gli operandi per poter essere eseguite velocemente, ma gli operandi salvati nella memoria richiedono tempi lunghi di accesso in memoria per poter essere recuperati. Per questo vengono definiti un numero limitato di **registri per memorizzare gli operandi più usati**. ARM ha 16 registri globalmente indicati come register file. **Meno sono i registri e più rapidamente sono accessibili, questa è un'applicazione del 3° principio più piccolo è più veloce. Infatti leggere un dato da pochi registri è molto più rapido che leggerlo da una memoria grande.** Il register file di solito è costituito da un array di memoria SRAM.

Applicazione del 4° principio ARM: Good design demands good compromises

ARM usa istruzioni da 32 bit. Siccome regolarità garantisce semplicità, la scelta che da la massima regolarità è quella di destinare **una parola di memoria a ciascuna istruzione**, anche se non tutte le istruzioni hanno bisogno di 32 bit per essere codificate. Usare istruzioni di lunghezza variabile vorrebbe dire aumentare inutilmente la complessità circuitale.

La semplicità suggerirebbe di avere un solo formato per le istruzioni, ma ciò sarebbe troppo restrittivo non si riesce in un unico formato a descrivere le particolarità di ogni istruzione: questo consente di introdurre il quarto principio ossia un buon progetto richiede buoni compromessi.

ARM sceglie il compromesso di avere 3 possibili formati di istruzione:

- **Istruzioni di Data-processing;** processano i dati memorizzati nei registri
- **Istruzioni di Memory;** che si dividono in istruzioni di LOAD o STORE
- **Istruzioni di Branch.** istruzioni di salto da un punto all'altro del programma

Il numero limitato di formati consente di avere una certa regolarità tra le varie istruzioni, quindi di semplificare i circuiti di decodifica pur soddisfacendo le esigenze di diverse istruzioni.

Cos'è un processore general purpose

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

Lo **stato (architetturale)** 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.

A partire dallo stato corrente il processore esegue una particolare istruzione su un particolare insieme di dati per produrre un nuovo stato architetturale.

Una microarchitettura quindi può essere vista come un automa

L'automa si trova inizialmente in un certo stato dato dal valore del clock, dai valori memorizzati nelle locazioni di memoria, nei registri, del program counter.

Quando viene eseguita una nuova istruzione che cambia il contenuto della memoria, dei registri ecc, si passa ad un nuovo stato dell'automa e così via per ogni istruzione.

Il numero di stati in cui può trovarsi una microarchitettura è esorbitante.

Variazioni di stato

I registri, la memoria dati e la memoria istruzioni operano mediante logica combinatoria.

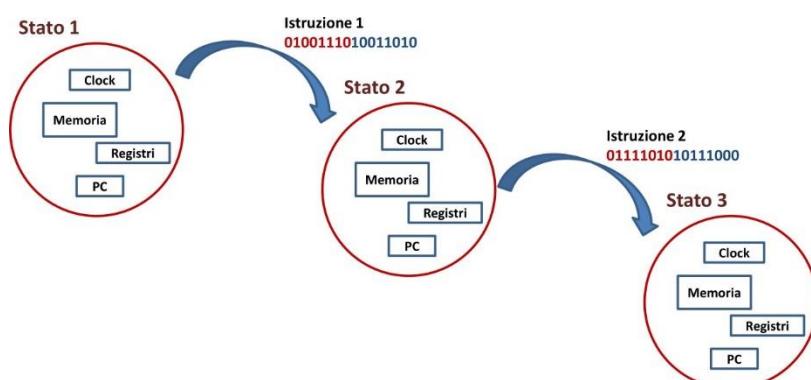
Le transizioni degli stati sono regolate da un clock che da sincronicità a tutto il sistema.

In particolare tutti i componenti effettuano la **scrittura sul fronte alto del clock**, cosicché **lo stato del sistema cambia solo su un fronte del clock (quello alto)**.

Per scrivere sui registri , memorie ecc. ci sono degli **abilitatori** che consentono di scrivere/leggere dai registri/memoria. Prima di eseguire una istruzione gli abilitatori devono essere impostati in maniera corretta.

Quindi gli indirizzi, i dati ed il segnale di write enable devono essere impostati prima del fronte alto del clock e mantenuti immutati per un tempo superiore al ritardo di propagazione.

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.



Tipologie di microarchitettture

Esistono 3 tipologie di microarchitetture per l'architettura ARM:

- **microarchitettura a ciclo singolo**
- **microarchitettura a ciclo multiplo (multiciclo)**
- **microarchitettura pipeline**

Esse differiscono su come vengono eseguite le istruzioni e per il modo in cui i vari elementi di stato sono connessi tra loro.

Microarchitetture a ciclo singolo

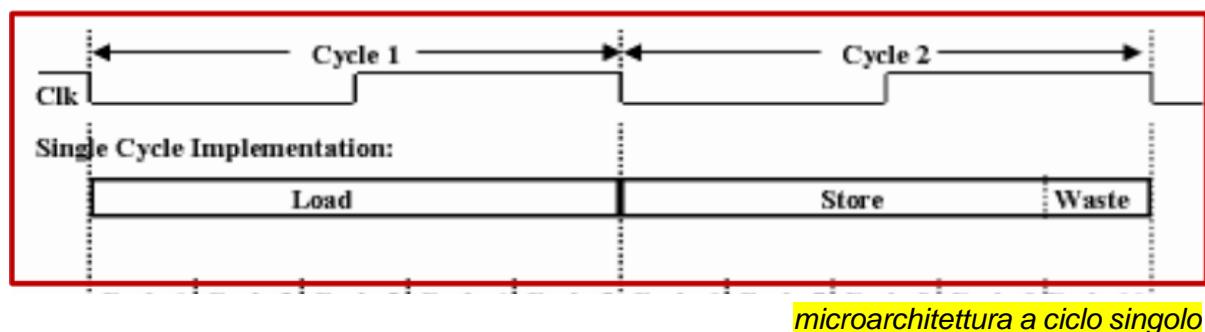
Sono le più semplici. **Un intera istruzione è eseguita in un singolo ciclo di clock.**

Vantaggi:

- semplice da comprendere;
- unità di controllo molto semplice;
- non richiede stati non architetturali (gli unici stati del sistema sono quelli dati dai valori correnti della memoria, registri ecc.)

Svantaggi:

- **periodo di clock pari a quella dell'istruzione più lenta;** (la frequenza o il periodo di clock del clock deve essere almeno pari a quello dell'istruzione più lenta)
- **memoria dati e memoria istruzioni separate;**



Microarchitetture a ciclo multiplo

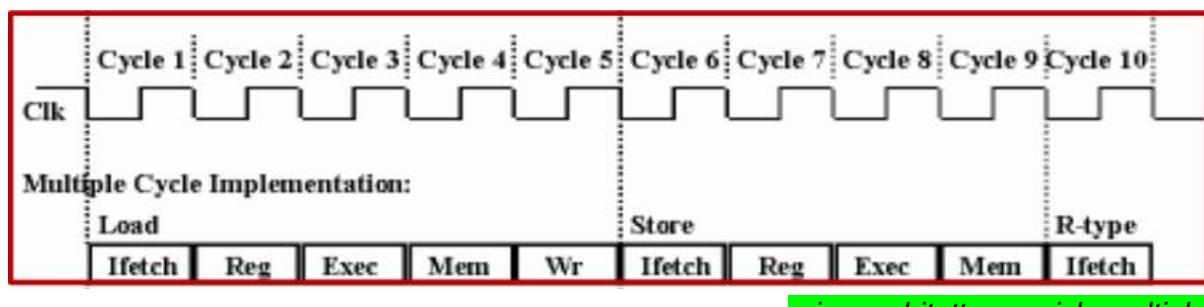
Un istruzione è eseguita in più cicli di breve durata.

Vantaggi:

- riuso dei componenti (riduce il costo hardware riutilizzando blocchi circuitali costosi come i sommatori e le memorie);
- durata variabile delle istruzioni (le istruzioni più semplici vengono eseguite in meno cicli di quelle più complesse);
- non richiede la separazione delle memorie

Svantaggi:

- richiede stati non architetturali
- esegue una sola istruzione alla volta ma ogni istruzione richiede più cicli di clock per essere completata



microarchitettura a ciclo multiplo

Microarchitetture con pipeline

Sono le più avanzate. **Eseguono più istruzioni contemporaneamente migliorando sensibilmente le prestazioni.**

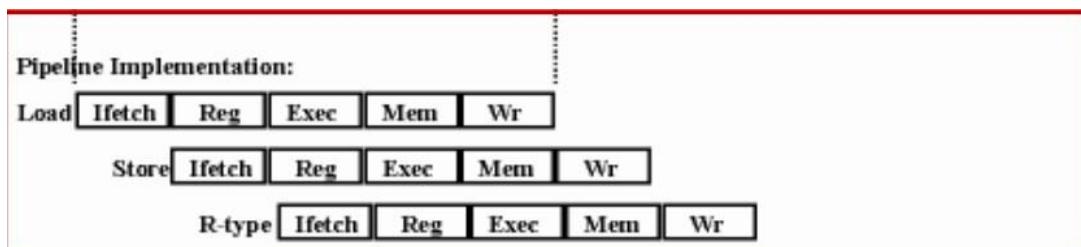
Si applica il concetto di pipelining al processore a ciclo singolo, migliorandone le performance. Ogni istruzione viene suddivisa in varie fasi e nel ciclo, mentre si esegue una fase di un'istruzione inizia anche una fase dell'istruzione successiva.

Vantaggi:

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

Svantaggi:

- logica di controllo più complessa
- richiede registri di pipeline



microarchitettura con pipeline

Misura delle prestazioni

Il processore può avere diverse microarchitetture con diversi rapporti costo/prestazioni.
Il costo dipende dalla quantità di hardware necessaria e dalla tecnologia di realizzazione.
Più ci sono porte logiche e memoria e più i costi sono maggiori.

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

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

In generale il tempo di esecuzione di un programma (in secondi) si calcola secondo la seguente formula:

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

Il numero di istruzioni in un programma dipende dall'architettura del processore, ma anche dall'abilità del programmatore.

Il numero di cicli per istruzione è detto CPI (Cycles Per Instruction) ed è il numero di cicli di clock richiesti in media per eseguire un'istruzione.

Il numero di secondi per ciclo è il periodo del clock (clock period) ed è determinato dal percorso critico attraverso i circuiti del processore. La frequenza di clock è l'inverso del periodo di clock ed è data dal numero di cicli per secondo (cycle/seconds).

L'inverso del CPI è la potenza di elaborazione (IPC Instruction Per Cycle)
(instructions/cycle)

Microarchitetture: schema generale

Un PROCESSORE è costituito da:

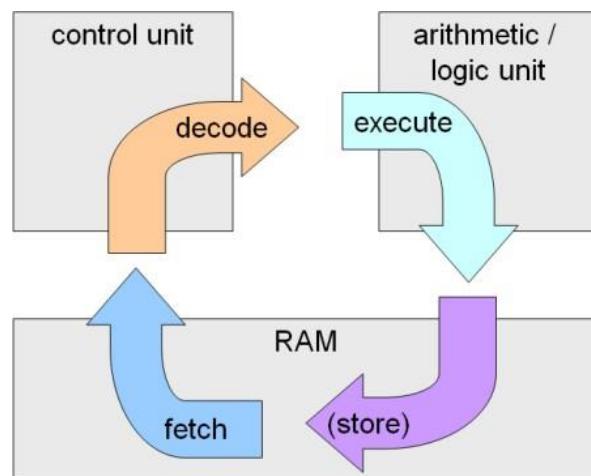
- un **datapath** ossia un circuito che realizza il flusso dei dati e l'esecuzione materiale dell'istruzione
- dalla **control unit** che istruisce come il datapath deve operare a seconda della tipologia d'istruzione

Una **istruzione** ha un **ciclo di vita** che consta di **4 fasi** principali: (fdes)

- 1) **fetch**
- 2) **decode**
- 3) **execute**
- 4) **store**

- 1) Nella fase di **FETCH** dell'istruzione, partendo da un registro che contiene l'indirizzo dell'istruzione corrente (cioè quella da eseguire) - PC - si va a ricercare in memoria tale istruzione e la si carica in un registro apposito (32 bit) dove ci sarà il contenuto di tale istruzione
- 2) Nella fase di **DECODE** dell'istruzione, eseguita da un'unità del processore chiamata **control unit CU**, essa decodifica l'istruzione e istruisce il circuito datapath su come deve operare a seconda della tipologia di istruzione da eseguire

- 3) Nella fase di **EXECUTE** dell'istruzione essa viene eseguita. Bisogna ricercare nei registri appositi gli operandi di tale istruzione , poi essa viene eseguita , calcolando il risultato mediante l'**ALU** che si occupa dei calcoli aritmetici e logici dell'istruzione
- 4) Nella fase di **STORE** dell'istruzione il risultato finale dell'istruzione viene memorizzato in memoria (ad esempio all'interno dei registri del file register che sono i registri presenti all'interno della cpu)



Progettazione di una microarchitettura ARM

Una microarchitettura consta di due componenti che **interagiscono** fra loro:

- **il datapath:** è un circuito che 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. E' a 32 bit.
- **l'unità di controllo:** riceve l'istruzione corrente dal datapath ed "istruisce" il datapath su come eseguire tale istruzione. "Istruisce" significa che definisce dei valori di selezione dei vari multiplexer del datapath e poi stabilisce i segnali di abilitazione di varie strutture quali le memorie, il file register.

L'unità di controllo riceve l'istruzione corrente dal datapath e comunica ad esso come eseguirla, attivando opportunamente gli ingressi di selezione dei multiplexer, le abilitazioni dei registri e i segnali di lettura e scrittura in memoria per controllare le operazioni del datapath.

I 5 componenti fondamentali di un processore (elementi di stato del datapath)

La memoria è suddivisa in 5 ELEMENTI DI STATO (pc, rf, sr, im, dm):

1-program counter

2-i registri (register file) ossia l'insieme di registri a 32 bit che corrispondono alla memoria operativa su cui l'alu opera

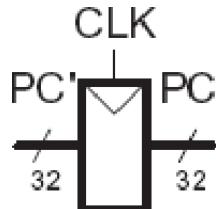
3-il registro di stato (status register) che è un registro a 4 bit

4-la memoria istruzioni (instruction memory)

5-la memoria dati (data memory)

A questi elementi di stato si aggiungono blocchi di logica combinatoria per generare il nuovo stato dell'automa a partire dallo stato corrente.

1. PROGRAM COUNTER



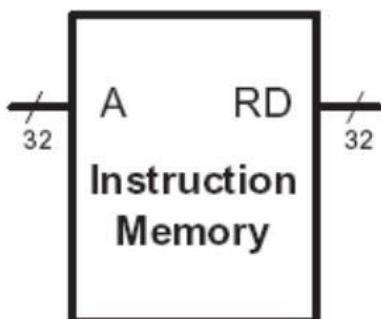
Il PC è un registro a 32 bit sincronizzato dal clock il cui scopo è quello di memorizzare di volta in volta l'istruzione corrente da eseguire. Ad ogni ciclo di clock conterrà l'istruzione che deve essere eseguita e poi si aggiornerà in modo tale che al prossimo clock conterrà l'istruzione successiva da eseguire.

Esso fa logicamente parte del register file, che contiene tutti i registri, ma viene letto e scritto ad ogni ciclo indipendentemente dagli altri registri ed è quindi implementato come un registro autonomo 32-bit.

La sua **uscita a 32 bit, PC, indica l'indirizzo dell'istruzione corrente.**

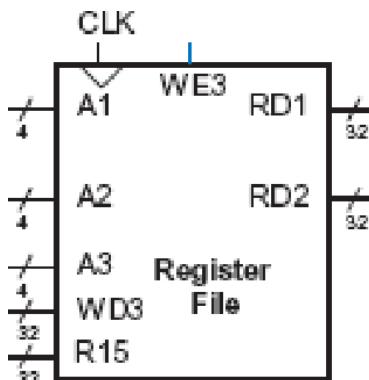
Il suo **ingresso, PC', indica l'indirizzo della successiva istruzione da eseguire.**

2. INSTRUCTION MEMORY



L' **instruction memory** è la memoria istruzioni. Essa ha una sola porta di lettura. Riceve in ingresso l'**indirizzo di un'istruzione a 32 bit, A**, ed emette sull'**uscita di lettura RD ReadData** (ossia un registro) il dato, appunto l'istruzione, contenuta nella parola di indirizzo A.

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? 2 alla 32 istruzioni cioè circa 4 miliardi di istruzioni



Il register file consiste di 16 registri (da R0 a R15) di 32 bit ciascuno

Dal punto di vista FISICO i registri sono equivalenti. Dal punto di vista LOGICO non sono equivalenti, svolgono al livello architetturale durante l'esecuzione di un programma funzioni particolari.

R13: registro SP STACK POINTER

R14: registro LR LINK REGISTER

R15: registro PC PROGRAM COUNTER (proveniente dal program counter, contiene l'indirizzo dell'istruzione successiva a quella contenuta nel PC vero e proprio)

L'ingresso di R15 è separato perché è collegato direttamente al PC da cui riceve l'indirizzo corrispondente alla prossima istruzione da eseguire e con il quale è sincronizzato

Vi sono diversi ingressi e uscite.

Ci sono due **porte di lettura A1 e A2** e una **porta di scrittura A3**.

Le **porte di lettura A1 e A2** ricevono in ingresso un input di 4 bit (2 alla 4=16), che specificano l'indirizzo di uno dei 16 registri come operando sorgente.

Il valore a 32 bit dei registri indirizzati viene letto sulle uscite di lettura dati **RD1 e RD2**.

Si utilizza evidentemente un MUX per selezionare il registro da cui leggere il valore a 32 bit, dati gli ingressi A1 e A2

Gli ingressi A3 e WD3 servono per memorizzare quindi "scrivere" un dato in un registro.

Costituiscono la porta di scrittura. Essa ha:

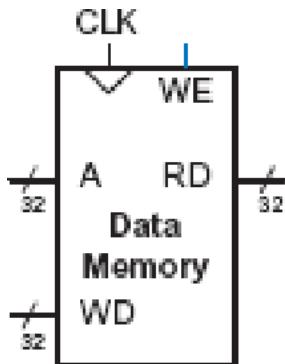
L'ingresso A3 che riceve un indirizzo di 4 bit che specifica il registro nel quale scrivere il dato.

L'ingresso WD3 che riceve il dato vero e proprio da memorizzare, a 32 bit.

Un **segnale di Write Enable**, di abilitazione alla scrittura che sarebbe **WE3**

Se il segnale di abilitazione alla scrittura **WE3 = 1** e quindi è attivo il dato specificato da WD3 viene memorizzato (scritto) nel registro specificato da A3 in corrispondenza del fronte di salita del clock.

4 DATA MEMORY



Il **Data Memory** ha una singola **porta di lettura/scrittura**, A, a 32 bit.

Il contenuto da scrivere in memoria viene dato all'ingresso **WD**.

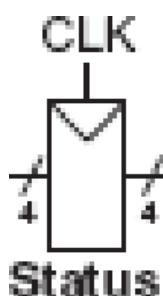
Quando il segnale **WE = 1** (Write Enable) è attivo, essa scrive la parola contenuta all'ingresso WD nella cella puntata dall'indirizzo A durante il fronte alto del clock.

Si prende l'indirizzo A, e si scrive nella cella di memoria corrispondente a tale indirizzo la parola che si trova in WD.

Quando il segnale **WE = 0** (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 in lettura.

Si prende l'indirizzo A, lo si decodifica, si legge il contenuto della cella di memoria e lo si riporta in RD.

5. STATUS REGISTER



Ha 4 ingressi in input usati per memorizzare lo stato dei flags **N,Z,C,V** forniti dall'ALU al fine di eseguire istruzioni condizionate.

Non sempre i flags in output dell'alu vengono memorizzati nello status register. solo per alcune istruzioni.

Questo registro permette di definire le **istruzioni condizionali** cioè istruzioni assembly che vengono eseguite solo se vengono soddisfatte alcune condizioni.

N=Negative

Z=Zero

C=Carry

V=Overflow

Istruzioni di base datapath

Studiamo il datapath di alcune istruzioni principali dell'architettura ARM, considerando un insieme limitato del set di istruzioni.

-**Istruzioni di data-processing:** ADD, SUB, AND, ORR (con registri e modalità di indirizzamento diretto, senza shift)

-**Istruzioni di memoria:** LDR, STR (con positive immediate offset)

-**Istruzioni di salto:** B

Indirizzi di memoria

La memoria è organizzata come un vettore di parole. L'architettura arm usa 32 bit per gli indirizzi di memoria e 32 bit per le parole (dati). La memoria si dice **byte-addressable** perché ad ogni byte di memoria (8 bit) corrisponde un unico indirizzo univoco.

Nell'architettura ARM operiamo con parole di 32 bit, che sono 4 byte (8 bit ciascuno).

Questo vuol dire che gli indirizzi delle parole incrementano di 4 in 4.

Il byte più significativo è a sinistra MSB (Most significant byte)

Il byte meno significativo è a destra LSB (Least significant byte)

Byte address				Word address
:				:
13 12 11 10				00000010
F E D C				0000000C
B A 9 8				00000008
7 6 5 4				00000004
3 2 1 0				00000000
MSB				LSB

Istruzioni di memoria: LOAD (LDR)

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]

In questo caso calcola un indirizzo a partire dal contenuto del registro R1 (**detto BASE ADDRESS**) sommato ad uno **spiazzamento (offset)** che è **dato da un immediate** (costante) ossia #12. Calcolato questo indirizzo preleva il contenuto locato in main memory a tale indirizzo e lo carica nel registro R0, **registro di destinazione**

R1 e l'immediate #12 sono operandi sorgente.

I dati locati in RAM devono essere caricati nei registri della cpu per eseguire le operazioni

Vi sono anche altre segnature di LDR, ad esempio:

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

R0 \leftarrow mem32[R1 + R2] R0 \leftarrow mem32[R1 + (R2*4)]

Nel primo caso vi è un offset da registro, non è indicata una costante come offset bensì un registr. In questo caso al contenuto del registro R1 che è il base address viene sommato un offset che è dato dal contenuto del registro R2.

Nel secondo caso abbiamo un offset da un registro shiftato.

Al contenuto del registro R1 che è il base address viene sommato un offset che è dato dal contenuto del registro R2 shiftato di 2 posizioni verso sinistra, il che vuol dire moltiplicare il contenuto del registro R2 per 4. ($R2^4$)

Quindi abbiamo 3 tipi di offset: da immediate, da registro, da registro shiftato

Istruzioni di memoria: STORE (STR)

Una istruzione che scrive in memoria è detta store

Mnemonico: STR (store register)

Semantica speculare a LDR

STR R0, [R1, #12]

mem32[R1 + 12] \leftarrow R0

Segniture analoghe

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

mem32[R0 + R2] \leftarrow R1 mem32[R1 + (R2*2)] \leftarrow R0

Prende il contenuto di un certo registro sorgente e lo salva in memoria ad un certo indirizzo
Il primo argomento è il registro che contiene la sorgente del dato che deve essere salvato in memoria ad un certo indirizzo.

Istruzioni di memoria: Varianti LDRB, STRB

LDRB/STRB eseguono rispettivamente il load e lo store di un **singolo byte** e non della parola a 32 bit (4 byte)

Organizzazione della memoria: come sono indirizzati i byte all'interno di una parola. Disposizione LITTLE ENDIAN e BIG ENDIAN

Le memoria byte addressable sono organizzate in modalità big-endian o little-endian, **che definiscono la numerazione dei byte in una parola.**

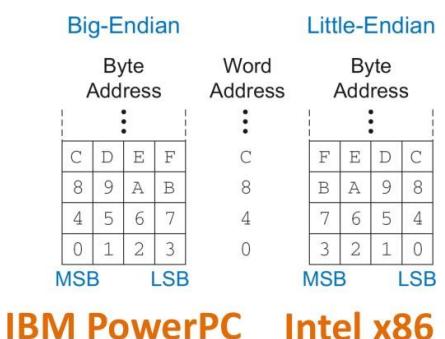
In entrambe il byte più significativo della parola MSB è a sinistra.

Il byte meno significativo della parola LSB è a destra.

little-endian: la numerazione dei byte inizia dal byte meno significativo LSB

big-endian: la numerazione dei byte inizia dal byte più significativo MSB

ARM non stabilisce se il tipo di indirizzamento dei byte in una parola deve essere little o big endian. Sceglie la casa produttrice del processore. Di solito si sceglie LITTLE ENDIAN.



Indicizzazione (tipi di indirizzamento)

ARM fornisce tre tipi di indicizzazione: **OFFSET, PREINDEX, POSTINDEX**

Offset: L'indirizzo è calcolato sommando o sottraendo un offset al contenuto del registro di base. Il registro di base non viene aggiornato ad un nuovo valore. **PW: 10**

Preindex: L'indirizzo viene calcolato come nel caso dell'offset, sommando o sottraendo un offset al contenuto del registro di base. La differenza è che il registro di base viene aggiornato col nuovo indirizzo calcolato. **PW: 00**

Postindex: L'indirizzo corrisponde al contenuto del registro di base.

Dopodiché l'offset viene aggiunto o sottratto al contenuto del registro di base che viene aggiornato col nuovo indirizzo appena calcolato con l'offset. **PW: 11**

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

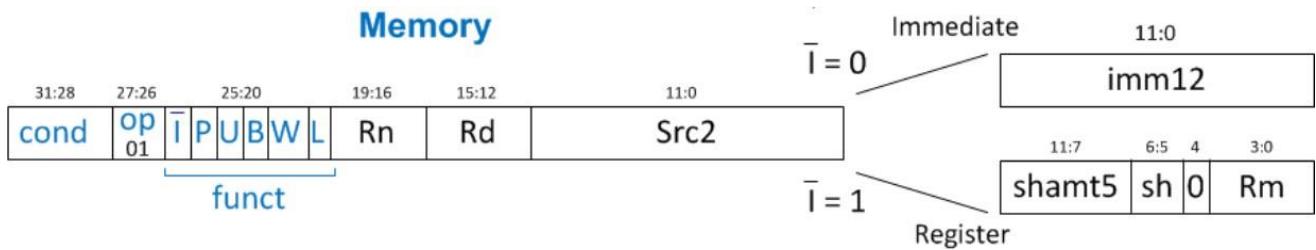
Offset: LDR R1, [R2, #4] ; R1 = mem[R2+4]

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

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

Codifiche istruzioni di memoria: LDR, STR, LDRB, STRB

Vengono codificate in parole a 32 bit.



cond: sono i **4 bit più significativi** dell'istruzione (28:31) e sono i bit di condizione, servono a codificare un'esecuzione condizionata ossia eventuali condizioni che determinano o meno l'esecuzione di questa istruzione.

Le istruzioni di memoria di solito sono **non condizionate** **cond = 1110 (14)**

op: sono **2 bit di operazione**. Siccome esistono 3 categorie di istruzione che hanno formati diversi, questi due bit specificano il formato di istruzione a cui ci si riferisce (istruzione di memoria, data processing o branch).

L'operation code per le istruzioni di memoria è **op = 01 (1)**

funct: sono **6 bit di controllo** ossia { I negato, P, U, B, W, L } corrispondono a significati diversi.

L : è il bit che ci dice se l'istruzione è di tipo LOAD o STORE.

Quando **L=0 abbiamo uno STORE**.

Quando **L=1 abbiamo un LOAD**.

B : è il bit che specifica se l'istruzione deve riferirsi ad una parola o un byte

Quando **B=0 si riferisce ad una PAROLA di memoria**

Quando **B=1 si riferisce ad un solo BYTE di memoria**

P e W : sono 2 bit che specificano il tipo di **INDIRIZZAMENTO** fra quelli possibili (**Postindex, Offset, Preindex**). La combinazione **01** è ridondante.

I negato : è 1 bit che indica se il **tipo di offset è da un immediate o un registro**. Indica come interpretare i 12 bit **Src2**

Quando **I negato=0** abbiamo un offset con **immediate (imm12)**

Quando **I negato=1** abbiamo un offset da **registro (Rm)**

U : è un bit che ci dice se l'offset va sommato o sottratto dal base address

Quando **U=0 l'offset viene sottratto** dal base address

Quando **U=1 l'offset viene sommato** al base address

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	T	U
0	Immediate offset in Src2	Subtract offset from base
1	Register offset in Src2	Add offset to base

Rn: sono 4 bit che specificano l'indirizzo del BASE ADDRESS, ossia il primo operando dell'istruzione di memoria, che è uno dei 16 registri (per questo 4 bit)

Rd: sono 4 bit che specificano l'indirizzo del primo argomento delle istruzioni di memoria.

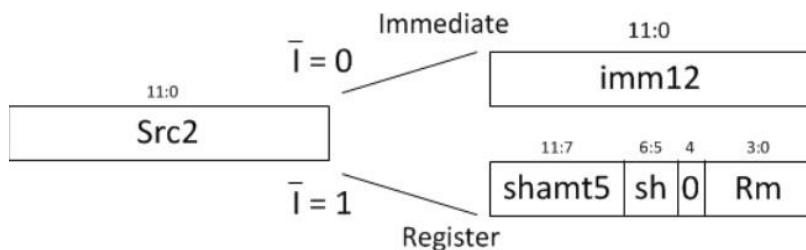
Se l'istruzione è di tipo **LDR (L=1)** questo è il **registro di destinazione**.

Se l'istruzione è di tipo **STR (L=0)** questo è il **registro sorgente** il cui contenuto viene memorizzato in memoria.

Src2: sono 12 bit che specificano il secondo operando delle istruzioni di memoria ossia l'offset. Questi 12 bit hanno due codifiche differenti a seconda del valore del bit di controllo **I negato**

Se **I negato=0** si interpretano i 12 bit di offset come un **immediate (costante) a 12 bit unsigned imm12**

Se **I negato=1** si interpretano i 12 bit di offset come un **offset preso da registro, eventualmente ruotato o shiftato di una certa quantità oppure shiftato di una quantità contenuta in un altro registro**.



Nel caso che **I negato = 1** l'offset è contenuto in un registro e il formato del campo Src2 cambia.

Abbiamo 4 campi: Rm, 0, sh, shamt5

Rm: sono 4 bit che specificano l'indirizzo del registro che contiene l'offset
Il bit 4 è sempre 0.

Sh: sono due bit che codificano il tipo di shift che si può avere, nel caso di un registro shiftato. **LSL 00 - LSR 01 - ASR 10 - ROR 11**

Non c'è un ASL perchè è sostituito da LSL equivalente.

ROL non c'è perchè possiamo ottenerlo sfruttando ROR. Il ROL sarà uguale al ROR 32-n

Infatti ruotare di n posizioni verso Sx significa ruotare verso destra di 32-n posizioni.

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

Shamt5: sono 5 bit che costituiscono lo “shift amount” cioè di quante posizioni deve shiftare il contenuto del registro Rm, fino ad un massimo di 32.

Datapath dell'istruzione LDR

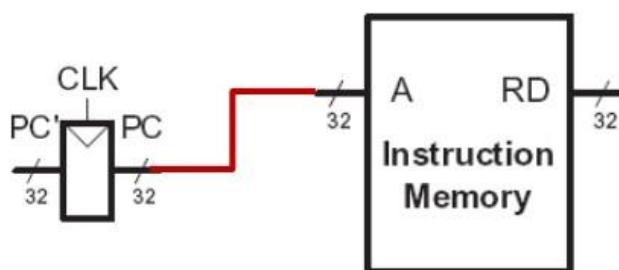
Il tipo dell'istruzione che consideriamo è con offset di tipo immediate (costante).

LDR Rd, [Rn, imm12]

Innanzitutto si parte dal **PC** che contiene l'indirizzo dell'istruzione da eseguire.

La prima fase è quella di **fetch dell'istruzione**, cioè si deve prelevare e leggere l'istruzione da eseguire che è contenuta nell' **instruction memory** all'indirizzo fornito dal PC.

Dunque il PC è collegato all'indirizzo di ingresso a 32 bit della memoria di istruzioni, **A**.

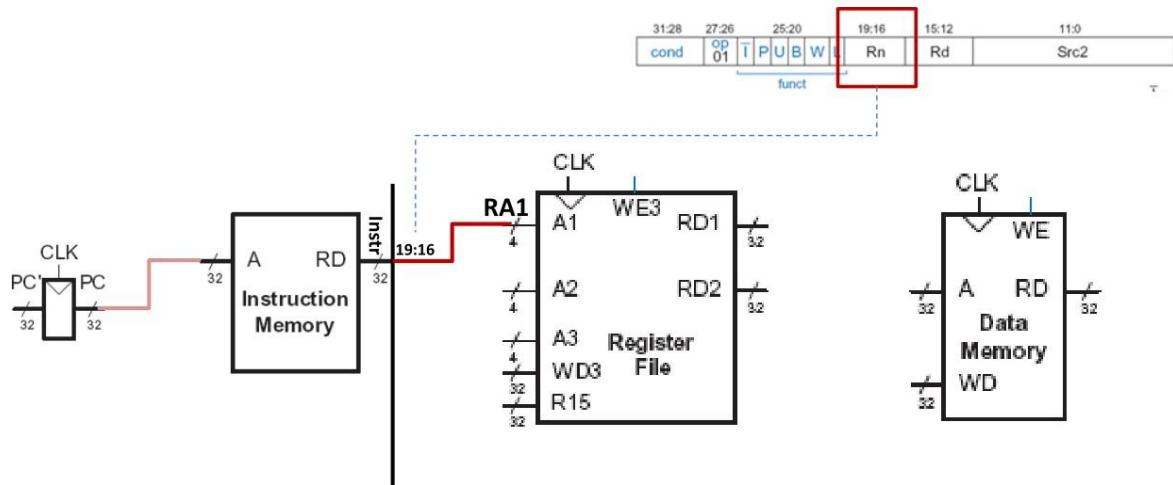


L'istruzione a 32 bit viene letta e riportata sull'uscita **RD** ed è rappresentata dall'etichetta **Instr**.

Per eseguire l'istruzione di LDR il passo successivo è quello di calcolare l'indirizzo da cui andare a prendere nella memoria dati (data memory) il dato da caricare all'indirizzo Rd. Tale indirizzo è calcolato sommando o sottraendo al valore Rn l'offset imm12 che è la costante.

Si inizia col prelevare il valore di **Rn** quindi si legge il **registro contenente l'indirizzo di base**. Il registro è specificato nel campo Rn della codifica dell'istruzione LDR, **Instr19:16 (4 bit)**

Questi 4 bit che specificano l'indirizzo del registro Rn vengono collegati all'ingresso di una delle porte del file register, **A1**. Il register file legge il contenuto del registro specificato ossia una parola a 32 bit e lo pone all'uscita **RD1..**



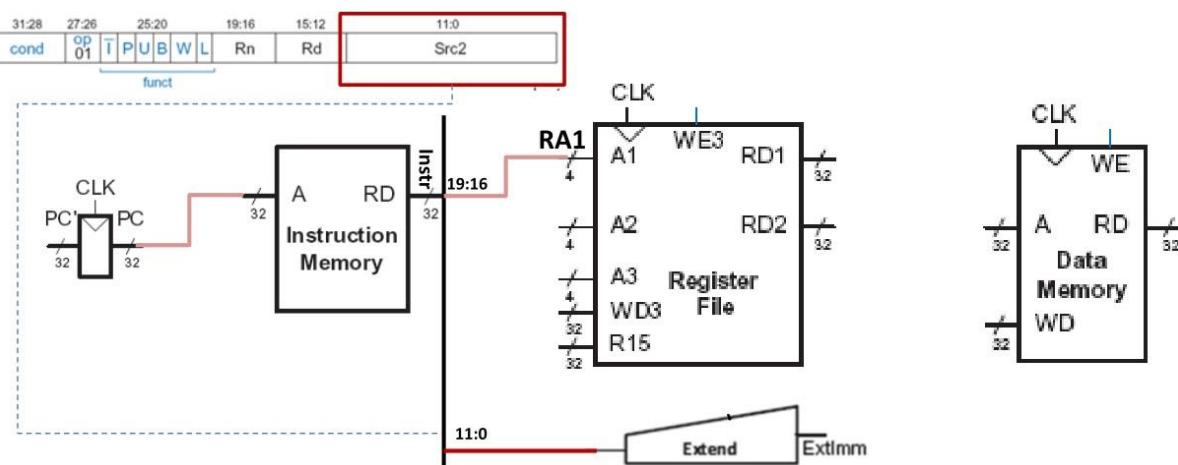
A questo punto ci occupiamo dell'**offset** che in questa istruzione di LDR è un immediate (costante) memorizzato nel campo **imm12** che sarebbero i primi 12 bit dell'istruzione

Istr11:0

L'offset è un valore unsigned a 12, che viene sommato o sottratto al valore Rn a seconda del bit U del campo di funct. Siccome Rn che è il base address ha un valore a 32 bit , per sommare l'offest dobbiamo **estendere il valore imm12 da 12 a 32 bit**.

I 12 bit imm12 vengono quindi prelevati dall'istruzione ed estesi a 32 bit attraverso un extender che produce il **valore esteso a 32 bit ExtImm**.

L'extender non fa altro che aggiungere degli zeri ai bit imm12 (Instr11:0) per farlo diventare a 32 bit. Quindi **ExtImm31:12=0 e ImmExt11:0=Instr11:0**



Adesso il processore deve sommare l'indirizzo di base Rn all'offset per ottenere l'indirizzo di memoria dati in cui andare a prelevare il dato da caricare nel registro di destinazione Rd. Per eseguire la somma si utilizza un ALU.

L'ALU riceve due operandi a 32 bit: **SrcA** e **SrcB**

- srcA proviene dal register file ossia dall'uscita RD1 e sarebbe il base address Rn
- srcB proviene da ExtImm e sarebbe il valore dell'offset imm12 esteso a 32 bit

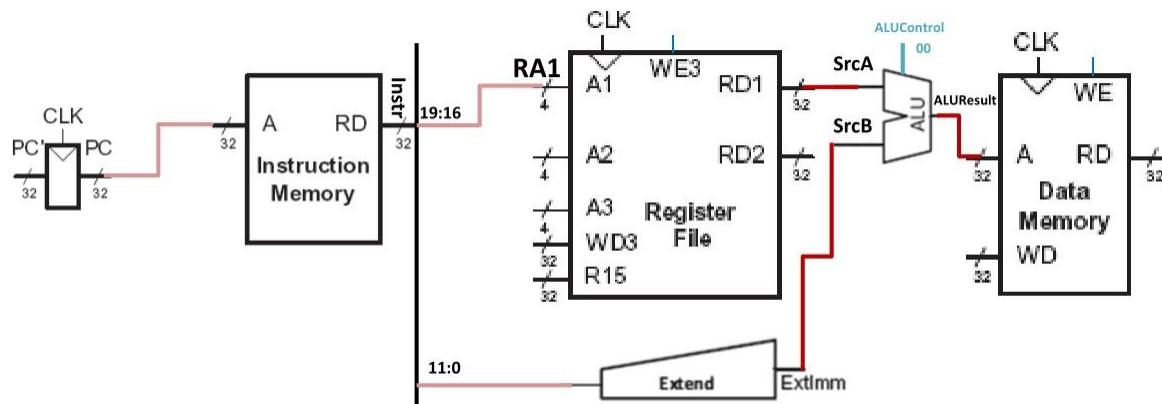
Sappiamo che l'ALU può effettuare varie operazioni aritmetiche/logiche. L'operazione da eseguire viene specificata dal segnale di controllo a 2 bit **ALUControl**.

ALUControl specifica quindi l'operazione da effettuare:

- una somma (ALUControl=00), se il valore del bit U = 1
 - una sottrazione (ALUControl=01), se il valore del bit U = 0

In questo caso siccome dobbiamo fare una somma tra Rn e offset (e quindi U=1) **ALUControl** vale **00** ed è impostato dalla CU.

La ALU dopo aver sommato i due operandi SrcA e SrcB genera un valore a 32 bit **ALUResult**. Questo risultato viene inviato alla memoria dati come indirizzo di lettura sulla porta **A**. Questo indirizzo è l'indirizzo in cui la memoria dati andrà a prendere il valore da caricare nel registro Rd.



Il dato viene prelevato dalla memoria dati e posto sull'uscita **RD** del data memory.

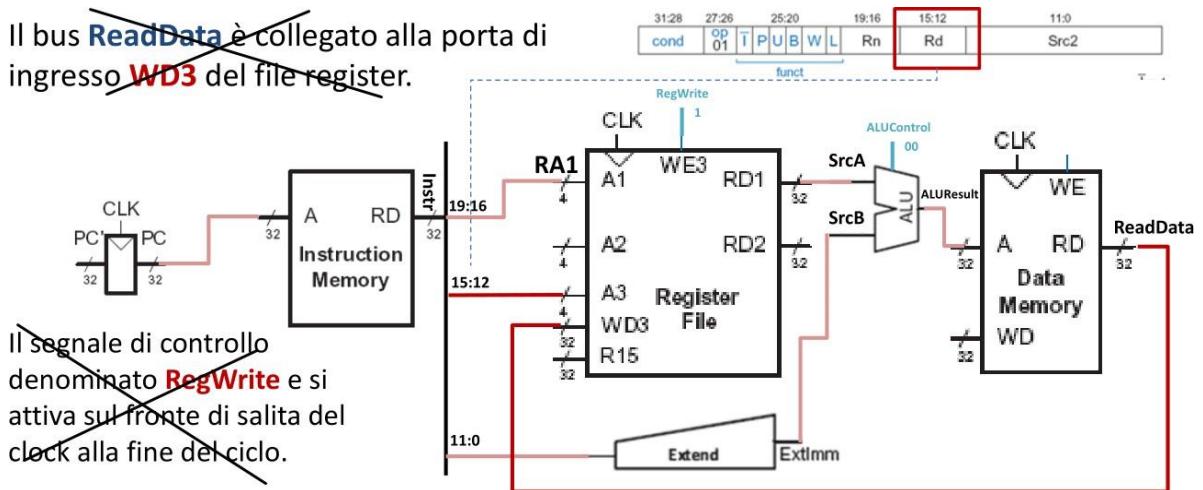
Esso poi passa sul bus **ReadData** e poi viene scritto nel registro destinazione Rd alla fine del ciclo, sul fronte alto del clock.

Il registro di destinazione per l'istruzione LDR è specificato nel campo **Rd** dell'istruzione **Instr15:12**. Questi 4 bit sono collegati all'ingresso **A3** del register file che sarebbe la porta di scrittura e questo ingresso A3 indica dove andare a scrivere il dato, appunto il registro Rd specificato dai 4 bit.

Il bus **ReadData** sul quale è presente il dato viene collegato alla porta di ingresso **WD3** del file register, che indica il contenuto del registro Rd specificato in A3.

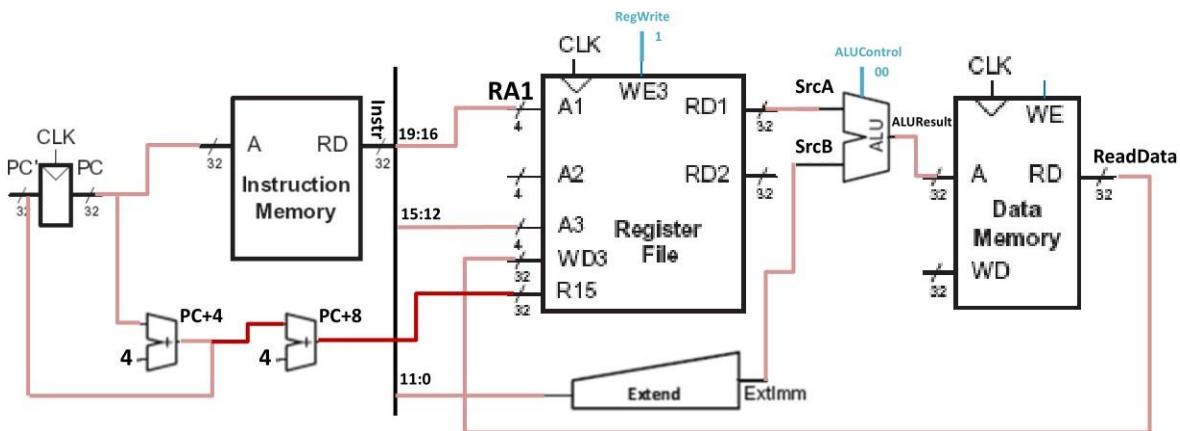
Il segnale di controllo denominato **RegWrite** è collegato all'abilitazione alla scrittura della porta A3, **WE3** e si attiva sul fronte di salita del clock alla fine del ciclo per scrivere il dato letto dalla memoria e contenuto in WD3 nel registro di destinazione Rd contenuto in A3.

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



Infine mentre l'istruzione viene eseguita , il processore deve calcolare l'indirizzo dell'istruzione successiva **PC'**. Siccome le istruzioni sono a 32 bit (4 byte), l'istruzione successiva è a **PC + 4**. Si utilizza un sommatore per incrementare il PC di 4. Il nuovo indirizzo viene scritto nel PC al successivo fronte di salita del clock.

Siccome nelle architetture ARM il registro **R15** contiene il valore **PC+8**, ossia contiene l'istruzione successiva a quella contenuta nel PC' ,è necessario un ulteriore **sommatore (+4)**, quindi **PC+4+4 = PC+8** la cui uscita è collegata all'ingresso di R15.



infine, se il registro di base o di destinazione è proprio R15 esso viene modificato e sovrascritto, e siccome deve contenere PC+8, deve mantenere una coerenza col PC, che quindi viene modificato .Quindi il valore del PC può provenire dal risultato dell'istruzione ReadData oppure da PC+4 PCPlus4 , si introduce un mux per scegliere fra le due possibilità **il multiplexer, permette di selezionare fra:**

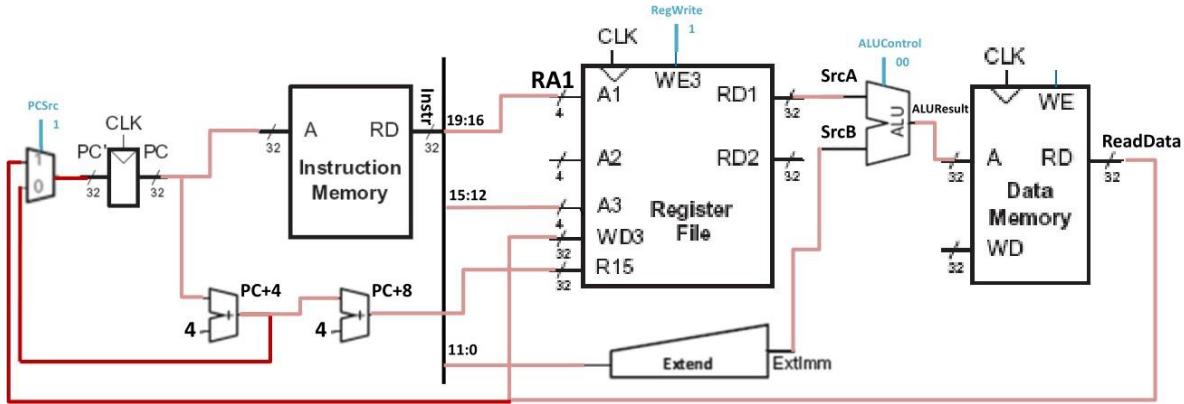
0- PCPlus4

1 – ReadData.

Il segnale di controllo associato al multiplexer è PCSrc.

Se PCSrc=0 seleziona PCPlus4

Se PCSrc=1 seleziona ReadData



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]

Si estende il datapath in modo da poter gestire anche le istruzioni di STR oltre che quelle di LDR viste in precedenza.

Per prima cosa si calcola l'indirizzo in cui dobbiamo andare a scrivere il dato contenuto in Rd. Questo calcolo è analogo a quello del LDR. Si prende l'indirizzo di base Rn e si somma all'offset imm12 che viene esteso a 32 bit.

L'indirizzo di base viene letto sempre dalla porta A1 del register file.

L'ALU aggiunge o sottrae a seconda del bit U l'indirizzo di base alla costante per trovare l'indirizzo di memoria nel quale memorizzare il dato.

Il dato che vogliamo salvare nel **data memory** tramite operazione di store si trova nel registro **Rd** specificato dai **4 bit Instr15:12 dell'istruzione di store**.

I 4 bit che specificano il registro Rd vengono collegati alla porta **A2** del register file.

Il **valore contenuto nel registro Rd** viene letto sulla porta di uscita del register file **RD2**.

La porta di uscita **RD2** del register file (che contiene il dato da salvare nella memoria dati) è collegata alla **porta di scrittura dati WD** del Data Memory.

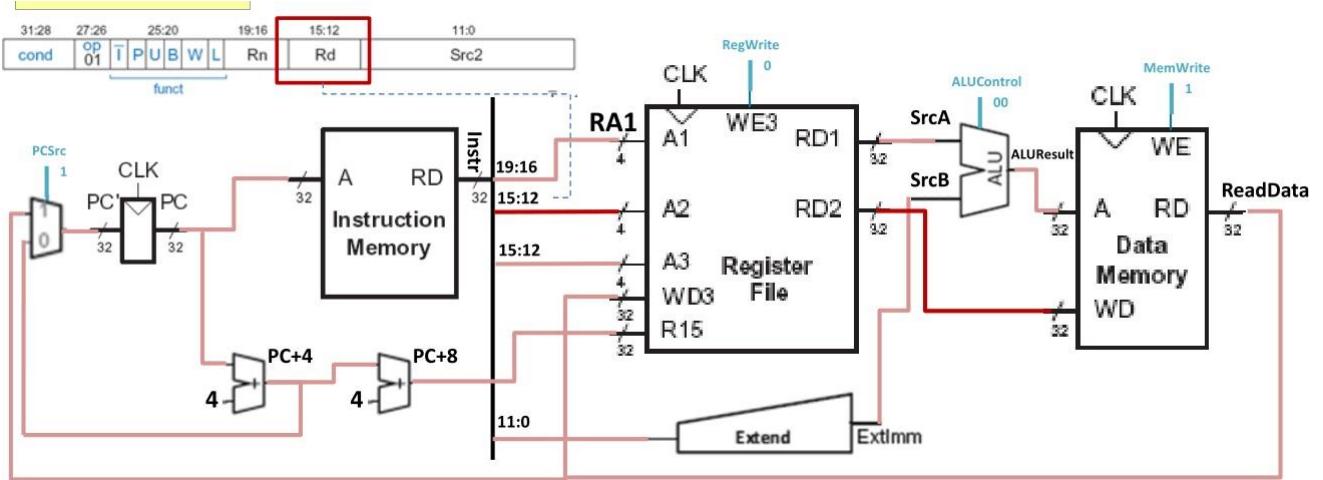
L'abilitazione alla scrittura in memoria dati **WE** è controllata dal segnale **MemWrite**

Quando MemWrite=1 il dato viene memorizzato in memoria

RegWrite=0 perché non c'è nessuna operazione di scrittura nel registro.

Pur essendo Rd associato anche a A3 e ReadData a WD3, questo non produce effetti sul File register, essendo RegWrite impostato a 0

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



Confronto RegWrite e MemWrite in operazioni di STR e LDR

Per quanto riguarda i segnali “RegWrite” e “MemWrite” nell’operazione di LDR

RegWrite=1 perchè dobbiamo scrivere nel registro

MemWrite=0 perchè l’accesso in memoria è solo in lettura per prelevare il dato

Nell’operazione di STR invece questi valori sono speculari perchè dobbiamo salvare il dato in memoria e quindi scrivere in memoria

MemWrite=1

RegWrite=0 perchè non ci serve scrivere nel register file

il PCSource in questo caso sarà uguale a 0, l’istruzione successiva PC’ ogni volta è calcolata come PC+4

Istruzioni di data processing

Sono istruzioni di elaborazione dei dati che eseguono appunto operazioni su dati memorizzati in registri e poi salvano i risultati delle operazioni in un registro di destinazione

MOV	Move a 32-bit value	MOV Rd, n	$Rd = n$
MVN	Move negated (logical NOT) 32-bit value	MVN Rd, n	$Rd = \sim n$
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	$Rd = Rn \text{ AND } n$
ORR	Bitwise OR of two 32-bit values	ORR Rd, Rn, n	$Rd = Rn \text{ OR } n$
EOR	Exclusive OR of two 32-bit values	EOR Rd, Rn, n	$Rd = Rn \text{ XOR } n$
BIC	Bit clear. Every '1' in second operand clears corresponding bit of first operand	BIC Rd, Rn, n	$Rd = Rn \text{ AND } (\text{NOT } n)$
CMP	Compare	CMP Rd, n	$Rd-n \text{ & change flags only}$
CMN	Compare Negative	CMN Rd, n	$Rd+n \text{ & change flags only}$
TST	Test for a bit in a 32-bit value	TST Rd, n	$Rd \text{ AND } n, \text{ change flags}$
TEQ	Test for equality	TEQ Rd, n	$Rd \text{ XOR } n, \text{ change flags}$
MUL	Multiply two 32-bit values	MUL Rd, Rm, Rs	$Rd = Rm*Rs$
MLA	Multiple and accumulate	MLA Rd, Rm, Rs, Rn	$Rd = (Rm*Rs) + Rn$

N. Matthiassen

MOV sposta nel registro di destinazione Rd un valore (costante o il contenuto di un registro),
ADD esegue una somma a 32 bit e salva il risultato nel registro di destinazione Rd
SUB sottrazione

Istruzioni per i branch

CMP esegue la sottrazione del primo operando col secondo e aggiorna i flags di stato NZCV. Memorizza nel registro current status program register i valori di uscita dei flags dell'operazione

CMN come CMP ma fa la somma anziché la sottrazione

TST fa l'and fra Rd e il secondo argomento e poi cambia i flags

TEQ fa lo xor fra Rd ed il secondo argomento e poi cambia i flags

Istruzioni di moltiplicazione

MUL moltiplica 32x32 bit Rs e Rm e mette il risultato a 32 bit in Rd

MLA moltiplicatore con accumulo fa la moltiplicazione tra Rm e Rs e poi somma un altro argomento Rn

UMULL moltiplicazione unsigned 32x32 bit, risultato a 64 bit salvato in 2 registri destinazione

SMULL moltiplicazione signed 32x32 bit risultato a 64 bit salvato in 2 registri destinazione

Istruzioni di data processing : istruzioni logiche

Operano bit a bit e scrivono il risultato in un registro di destinazione. La prima sorgente è sempre un registro e la seconda può essere un immediate o un altro registro

- **AND**: esegue l'and bit a bit
- **ORR**: esegue l'or bit a bit
- **EOR (XOR)**: exclusive OR ossia XOR bit a bit
- **BIC (Bit Clear)**: ogni volta che c'è un 1 nel secondo operando il valore del risultato è 0 maschera i bit
- **MVN (MoVe and NOT)**: commuta il valore del secondo operando e lo sposta in Rd

Le istruzioni AND or BIC sono utili per mascherare bit .

L'istruzione ORR: è utile per combinare bit fields

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

Assembly code		Result			
AND R3, R1, R2	R3	0100 0110	1010 0001	0000 0000	0000 0000
ORR R4, R1, R2	R4	1111 1111	1111 1111	1111 0001	1011 0111
EOR R5, R1, R2	R5	1011 1001	0101 1110	1111 0001	1011 0111
BIC R6, R1, R2	R6	0000 0000	0000 0000	1111 0001	1011 0111
MVN R7, R2	R7	0000 0000	0000 0000	1111 1111	1111 1111

Istruzioni di data processing : istruzioni di shifting

Le istruzioni di shift traslano a sinistra o a destra il valore contenuto in un registro eliminando i bit a una delle due estremità. L'istruzione di rotazione invece fa ruotare il valore di un registro fino a un massimo di 31 posizioni.

Sono 4 perchè la arithmetic shift left ASL è analogo al LSL e perchè il ROL viene implementato attraverso un ROR di 32-n posizioni.

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 2^n .

Analogamente, uno **shift aritmetico aritmetico a destra di un valore N** è equivalente a **dividere per 2^n** .

Sono:

- LSL logical shift left 00 => inseriscono zeri nei bit meno significativi a dx lasciati liberi
- LSR logical shift right 01 => zeri inseriti nei bit più significativi
- ASR arithmetic shift right 10 => si inseriscono bit pari al bit più significativo
- ROR rotate right 11

L'ampiezza della traslazione può essere definita da una costante o dal contenuto in un registro

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
R6	0000 0000	0000 0000	0000 0000
1110 0111			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

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**, eventualmente **traslato o ruotato**.

La **destinazione** è un **registro**.

L'istruzione è a 32 bit. Formato simile alle istruzioni di memoria: 2° principio regularity supports design simplicity

Data-processing

cond	op	funct	Rn	Rd	Src2
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

Abbiamo 6 campi: **cond, op, funct, Rn, Rd, Src2**

Gli **operandi** dell'istruzione sono codificati in 3 campi:

- **Rn**: campo a 4 bit che specifica il registro che contiene il valore del primo operando sorgente
- **Src2**: campo a 12 bit che specifica il registro (Shiftato, ruotato) o l'immediate che indicano il secondo operando sorgente

- **Rd**: campo a 4 bit che specifica il registro destinazione in cui viene salvato il risultato dell'istruzione di data processing

Campi di controllo:

cond: sono i **4 bit più significativi** dell'istruzione (28:31) e sono i bit di condizione, servono a codificare un'esecuzione condizionata ossia eventuali condizioni che determinano o meno l'esecuzione di questa istruzione. **non condizionate cond = 1110 (14)**
cmd = 11012 (13) codice usato per tutti i tipi di shift (LSL, LSR, ASR, ROR)

op: **operation code**, sono **2 bit** che specificano l'istruzione da eseguire. Per le istruzioni di data processing il suo valore è **00**

funct: sono **6 bit** che specificano il tipo di funzione da eseguire. Questo campo è formato da 3 sottocampi: **I, cmd, S**

I : è **1 bit** che serve a interpretare i 12 bit del campo Src2.

Se **I=0** , Src2 è un registro

Se **I=1** , Src2 è un immediate

cmd : sono **4 bit** che specificano l'**istruzione di data processing specifica da svolgere**.

Ad esempio

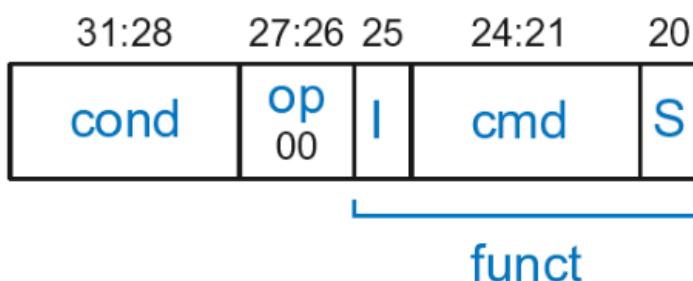
cmd=0100=ADD

cmd=0010=SUB

S-bit: è **1 bit** che serve per impostare ed aggiornare le **conditions flags NZCV** dell'alu al termine dell'istruzione.

Quando **S=1** vengono aggiornate le condition flags

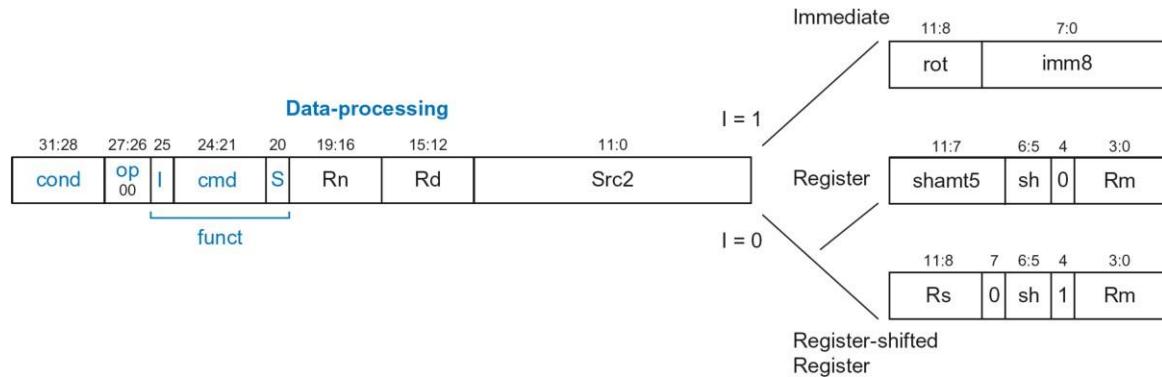
Quando **S=0** NON vengono aggiornate le condition flags



Campo Src2

Per quanto riguarda il campo **Src2** esso può essere a seconda del **bit I**:

- **un Immediate**
 - **un Registro** eventualmente traslato di una costante shift amount, oppure può essere anche un Registro "shiftato" dal contenuto di un altro registro



Nel caso **I=1** abbiamo che il campo a 12 bit **Src2** è un **immediate**.

L'immediate è codificato con **12 bit** e ha 2 campi: **rot e imm8**

rot: sono **4 bit** che specificano il **rotation value**, ossia sono dei bit che ci dicono di quanto deve **ruotare la costante imm8 verso destra**, questo ci consente di ottenere una costante dal valore molto alto quando essa viene estesa a **32 bit** (infatti $\text{rot} \times 2$, se $\text{rot}=1111=16$, $\text{rot} \times 2=32$) **imm8 ROR (rot × 2)**

Es. per produrre 0xFF0 imm8 deve essere ruotato di 4 bit a sinistra, ovvero 28 bit a destra. Quindi, rot = 14 , dato che rot * 2 = 28

imm8: sono 8 bit che specificano un unsigned immediate

Nel caso **I=0** abbiamo che il campo a 12 bit **Src2** è un **registro**.

Rm: sono **4 bit** che indicano il registro che fa da secondo operando

sh: **2 bit** che indicano il tipo **tipo di shift LSL 00 - LSR 01 - ASR 10 - ROR 11**

bit 4 : è quello che ci dice come interpretare i 5 bit da 11:7 per capire se il **valore di shift** è una **costante** oppure è contenuto in un **registro**.

Se il bit 4=0 : gli ultimi **5 bit** codificano il valore di shift il quale è indicato da una **costante shamt5**, shift amount che indica di quanto il valore in Rm è shiftato

Se il bit 4=1: il bit 7 rimane a 0 e i successivi 4 bit rimanenti indicano il registro **Rs** ossia l'indirizzo del registro nel quale è contenuto un valore che indica di quanto shiftare Rm. il numero di posizioni di cui Rm deve shiftare non è una costante ma è indicato dal registro **Rs**.

Datapath per le istruzioni di data processing di tipo logico: ADD, SUB, AND, ORR

Estendiamo il datapath per gestire le istruzioni di data processing ADD, SUB, AND e ORR.

Si considera inizialmente un'istruzione il cui secondo operando è un immediate, quindi un'istruzione con indirizzamento immediato. es. **SUB R2, R3, #0xFF0**

In tal caso, le istruzioni hanno come operandi un **registro Rn** ed una **costante imm8** contenuta nei bit dell'istruzione stessa. E' l'ALU esegue l'operazione di data processing logica specificata dall'istruzione e il risultato viene scritto in un registro di destinazione **Rd**

Queste istruzioni di data processing differiscono solo nella specifica operazione eseguita dall'ALU. Quindi, possono essere implementate tutte con lo stesso hardware utilizzando diversi segnali **ALUControl** che specificano l'istruzione logica da eseguire. La **CU si occupa di settare opportunamente il segnale ALUControl** a seconda dell'istruzione logica da eseguire.

I valori per **ALUControl** sono:

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

L'ALU al termine dei calcoli da effettuare genera anche **4 flags, ALUFlags3:0 (1 bit per flag, Negative Zero Carry oVerflow) che vengono inviati alla CU.**

A differenza delle istruzioni di memoria LDR e STR dove il campo Src2 dell'istruzione è a 12 bit se è l'indirizzamento è di tipo immediate e che viene esteso tramite l'extender a 32 bit, **nelle istruzioni di data processing il campo Src2 se è di tipo immediate è di 8 bit**

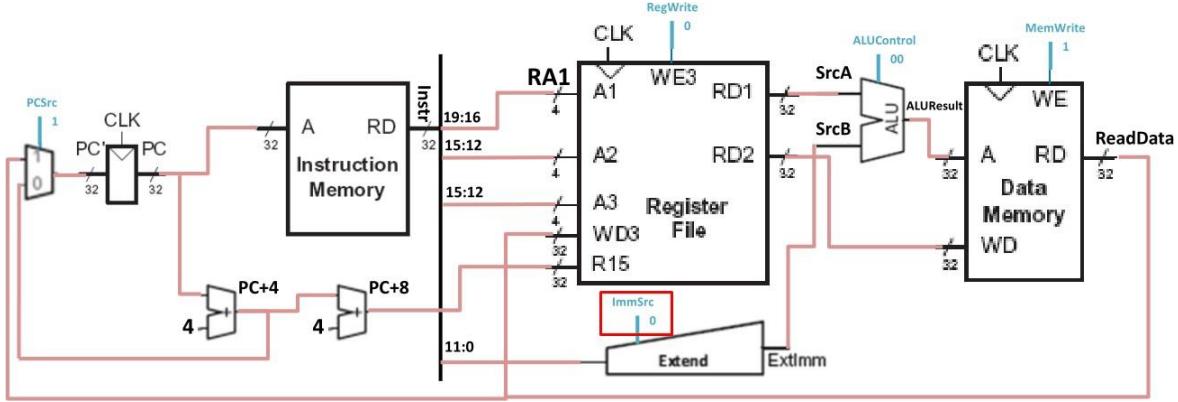
Quindi le istruzioni di data processing utilizzano costanti di 8 bit (non 12 bit). Questo vuol dire che l'extender deve funzionare in maniera diversa a seconda che l'istruzione sia di data processing o memoria.

Per cui il blocco circuitale **Extend** riceve in input un **segnale di controllo ImmSrc** :

**Se ImmSrc = 0 → ExtImm cioè l'immediate è esteso da Instr7:0
(a partire dall'8 bit vengono aggiunti gli 0, quindi si tratta di un immediate riferito a istruzioni di data processing)**

**Se ImmSrc = 1 → ExtImm cioè l'immediate è esteso da Instr11:0
(a partire dal 12esimo bit vengono aggiunti gli 0, quindi si tratta di un immediate riferito a istruzioni di memoria LDR STR)**

Il segnale ImmSrc è pilotato sempre della CU a seconda dell'istruzione che si sta eseguendo.



Un altro aspetto da disambiguare riguarda la scrittura nel register file.

L'ALU dopo aver effettuato l'operazione tra SrcA (registro) e SrcB (immediate) genera un risultato: **ALUResult** che deve essere memorizzato nel registro destinazione corrispondente Rd che si trova nel Register File.

Il register file quindi riceve il risultato dell'operazione in input sulla porta WD3:

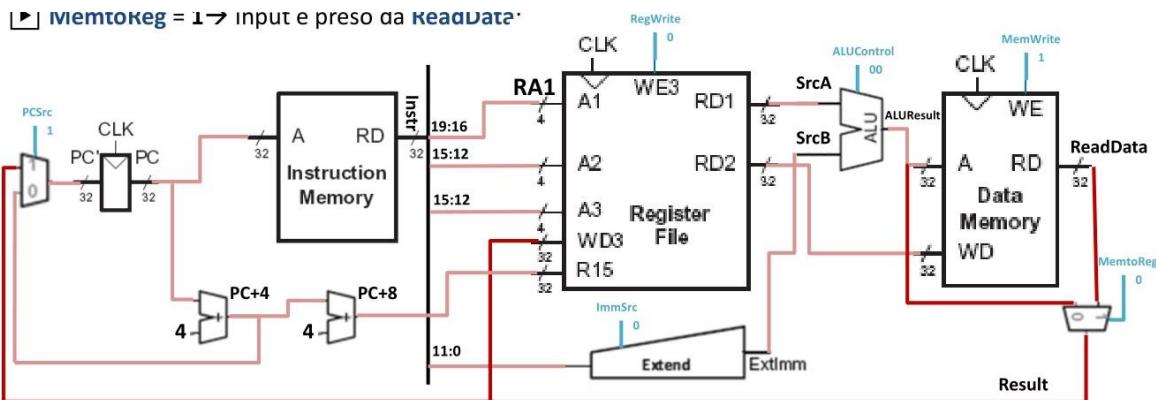
- sia dal data memory attraverso il bus **ReadData** in caso di istruzione di LDR
- sia direttamente dall'ALU nel caso di operazioni aritmetiche tramite **ALUResult**

Serve quindi un ulteriore multiplexer per selezionare con un selezionatore il risultato dell'operazione tra ReadData e ALUResult a seconda dell'istruzione. L'uscita di tale mux è denominata **Result** ed è collegata all'ingresso WD3 della porta di scrittura del RegisterFile.

Il multiplexer richiede un segnale di controllo per selezionare l'input corretto, ovvero **MemtoReg.**

Se MemtoReg = 0 → input è preso da ALUResult e trasferito al mux
(l'istruzione è di data processing);

Se MemtoReg = 1 → input è preso da ReadData e trasferito al mux
(l'istruzione è di memoria di tipo LDR, per quella STR il valore non interessa dato che non scrive in memoria);



Consideriamo ora le istruzioni di data processing con indirizzamento da registro es. EOR R8, R9, R10, ROR R12

esse ricevono il secondo operando sorgente da Rm, ossia il registro specificato dai primi 4 bit meno significativi dell'istruzione Instr3:0 quando I=0

Rm indica l'indirizzo del registro in cui andare a prendere nel register file il valore del secondo operando e quindi si deve poter associare in questo caso i 4 bit di Rm all'ingresso A2 del register file.

L'ingresso A2 del register file è associato però anche ai bit 15:12 che specificano Rd nel caso di istruzione di memoria di tipo STR.

Si aggiunge allora un ulteriore mux sull'ingresso A2 del file register.

Esso è pilotato dal segnale di controllo RegSrc . In base al suo valore stabilito dalla CU

=> A2 riceve in ingresso Rd (Instr15:12) (istruzione di STR)

=> A2 riceve in ingresso Rm (Instr3:0) per istruzioni di data processing con indirizzamento da registro

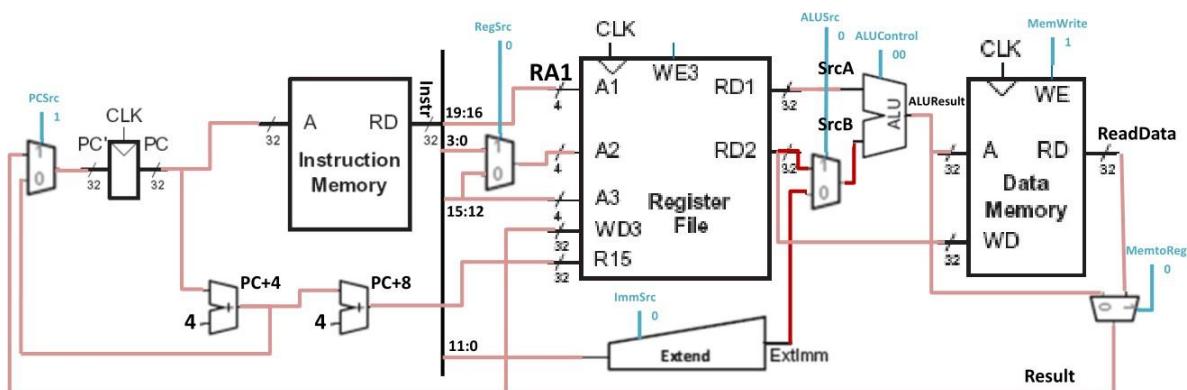
Infine nelle istruzioni di data processing il secondo operando SrcB in ingresso all'ALU può essere quindi un immediate a 8 bit che viene esteso a 32 bit e portato in ingresso all'ALU oppure un valore contenuto nel registro Rm che viene prelevato dal register file.

Quindi aggiungiamo un ulteriore multiplexer sugli ingressi dell'ALU per selezionare questo secondo operando sorgente.

E' pilotato dal segnale di controllo ALUSrc, che seleziona la seconda sorgente della ALU SrcB.

A seconda del segnale di controllo ALUSrc , SrcB viene selezionato da:

- ExtImm ossia un immediate esteso a 32 bit per istruzioni, che utilizzano costanti;
- dal register file sull'uscita RD2 per istruzioni di data processing con indirizzamento da registro.



Esecuzione condizionata di un'istruzione

Come abbiamo visto nei formati delle istruzioni precedenti i primi 4 bit più significativi costituiscono i bit di condizione. Un'esecuzione non condizionata ha il codice 14 (1110). ciò indica che l'istruzione è eseguita sempre e in modo incondizionato dai valori del CURRENT PROGRAM STATUS REGISTER (cpsr) . **A volte si vuole che l'esecuzione di una istruzione dipenda dal verificarsi o meno di una certa condizione. Sono istruzioni che vengono eseguite solo se certe condizioni dei 4 flags N Z C V memorizzati nel CPSR vengono soddisfatte. Per questo si dicono istruzioni condizionate.** Queste istruzioni di basso livello assembly consentono a linguaggi di programmazione di alto livello di realizzare costrutti di selezione quali if..else oppure iterazione while, for.

Il suffisso -S alle istruzioni di data processing

I flag N Z C V sono dette **CONDITION FLAGS** e sono memorizzate nel CPSR.

Questi flag sono settati da istruzioni di data processing a cui viene aggiunta una S finale (ad esempio ADDS) . La S finale indica che una volta eseguita l'istruzione i valori dei flag N Z C V di output dell'alu devono essere memorizzati nel CPSR . Può essere aggiunto a tutte le istruzioni di data processing

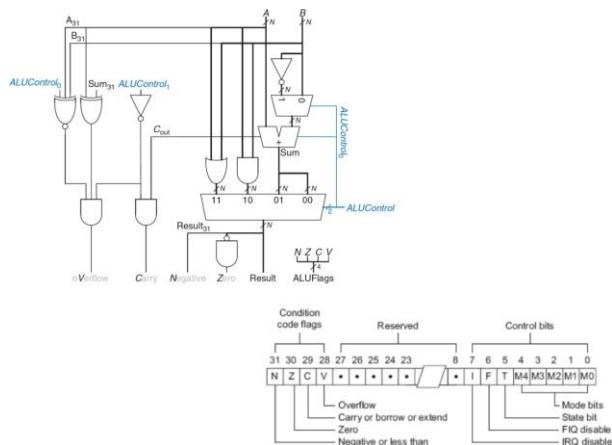
Esempio ADDS R1, R2, R3

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

Il CPSR

I valori dei flag N Z C V vengono dall'ALU (non sempre vengono memorizzati nel CPSR) . Sono memorizzati nei 4 bit più significativi del CURRENT PROGRAM STATUS REGISTER, che è un registro a 32 bit che presenta altri bit oltre questi 4 per gestire funzioni del s.o. come context switch o interrupt.



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

Istruzioni condizionate

Le istruzioni condizionate in assembly sono un particolare tipo di istruzioni che sono definite attraverso lo mnemonic che indica il tipo di operazione da eseguire (tipo ADD) al quale viene aggiunto un suffisso che indica la condizione che deve essere soddisfatta affinché l'istruzione e quindi l'operazione venga eseguita.

L'istruzione sarà eseguita condizionalmente sulla base dei valori dei condition flags.

Il suffisso dell'istruzione condizionata è detto **condition mnemonic**

Ogni condition mnemonic corrisponde ad una particolare condizione e ad un particolare stato in cui devono trovarsi i flag N Z C V per far sì che l'istruzione venga eseguita

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.

Es.

CMP R1, R2

SUB**NE** R3, R5, R8

- **NE:** SUB verrà eseguito solo se $R1 \neq R2$, ovvero $Z = 0$. Se Z fosse uguale a 1 vuol dire che i valori contenuti in R1 e R2 sono uguali , la sottrazione col CMP ritorna 0 e imposta il flag $Z=1$ e quindi not equal non è soddisfatta quindi non esegue l'istruzione

Tabella dei condition mnemonic

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

(a) $A = 1001_2 \quad A = 9 \quad A = -7$
 $B = 0010_2 \quad B = 2 \quad B = 2$
 $A - B: \begin{array}{r} 1001 \\ + 1110 \\ \hline 10111 \end{array} \quad NZCV = 0011_2$
 $\qquad\qquad\qquad HS: \text{TRUE} \quad GE: \text{FALSE}$

(b) $A = 0101_2 \quad A = 5 \quad A = 5$
 $B = 1101_2 \quad B = 13 \quad B = -3$
 $A - B: \begin{array}{r} 0101 \\ + 0011 \\ \hline 1000 \end{array} \quad NZCV = 1001_2$
 $\qquad\qquad\qquad HS: \text{FALSE} \quad GE: \text{TRUE}$

HS controlla bit C = true per essere eseguita

GE controlla la condizione *(N xor V) = true per essere eseguita

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 assume 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 le istruzioni in sequenza, incrementando il Program Counter (PC) di 4 byte (32 bit) dopo ciascuna istruzione, in modo da puntare alla successiva istruzione da eseguire.

Ci possono però essere delle eccezioni, ad esempio nelle istruzioni di data processing quando il registro di destinazione è proprio R15, che contiene l'istruzione successiva a quella contenuta nel PC e questo vuol dire che dobbiamo modificare di conseguenza anche il PC.

Le istruzioni di branching permettono di modificare il flusso e l'esecuzione delle istruzioni, non rendendole più sequenziali. Esse permettono di compiere dei salti da un punto all'altro del programma. **Modificano quindi attraverso i salti (branch) il flusso dell'esecuzione del programma**

Le istruzioni branching permettono di cambiare il valore del PC.

ARM include 2 tipi di branch:

- simple branch (B)
- branch and link (BL).

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

spesso sono condizionati cioè si verificano ad una certa condizione espressa da un suffisso e permettono di implementare a basso livello cicli ecc.

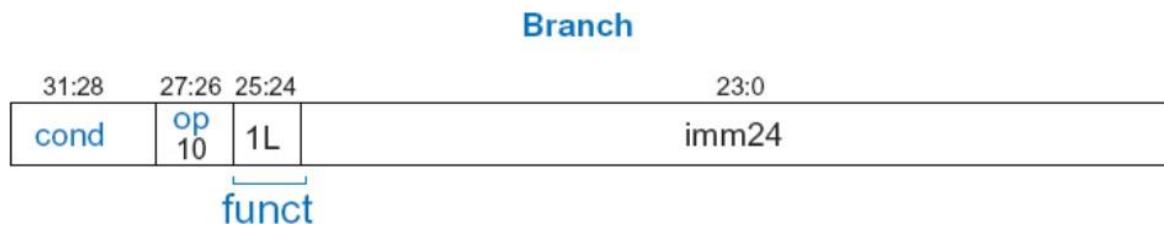
Il branch semplice (B) è utilizzato per realizzare istruzioni di controllo

Il branch and link (BL) viene utilizzato per le chiamate a funzione

Quando c'è una chiamata a funzione c'è un salto da un punto all'altro del programma, un cambio del flusso di controllo. dobbiamo salvarci l'indirizzo dell'istruzione seguente a quella di branch per ritornare nella funzione chiamante e ripartire col blocco di istruzioni successive al branch dopo aver eseguito quelle della subroutine.

Il codice assembly utilizza le etichette per indicare i blocchi di istruzione nel programma , blocchi che si possono saltare o ripetere. Quando il codice assembly è tradotto in linguaggio macchina queste etichette vengono tradotte in indirizzi di istruzione

Formato delle istruzioni di branching



Le istruzioni di branching utilizzano un **unico operando costante di 24 bit, imm24**. Questa costante è rappresentata in complemento a due (per avere salti maggiori) e specifica l'indirizzo dell'istruzione alla quale saltare, partendo dal valore PC+8.

cond: sono i primi 4 bit significativi che indicano la condizione

op: campo di 2 bit, che specifica l'operazione il cui valore è **10**. (istruzione di branch)

funct: il campo di funct è formato da 2 bit.

Il primo bit è sempre 1

Il secondo bit **L** assume il valore:

- 0 se il branch è semplice (B)
 - 1 se è un branch and link (BL)

La costante a 24-bit viene moltiplicata per 4 per sapere di quante parole saltare ed estesa con segno a 32 bit.

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 (memorizzato in R5) e scrive il risultato di nuovo nel PC.

Extender deve effettuare 3 forme di estensione diversa a seconda di ImmSrc

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 $Instr_{23}$ } $Instr_{23:0}$	24-bit signed immediate multiplied by 4 for B

00=>8 bit estesi a 32 per istruzioni di data processing

01=>12 bit estesi a 32 per istruzioni di memoria

10=>24 bit estesi a 32 (con segno) moltiplicati per 4 per istruzioni di branch

Istruzione BL

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

Essa salva l'indirizzo di ritorno, cioè l'indirizzo che è memorizzato nel registro R15 (PC), nel registro R14 (LR). Poi salta alla prima istruzione della subroutine ed il ritorno dalla subroutine una volta eseguita avviene ripristinando (cioè copiando) l'indirizzo contenuto nel registro R14 in R15 quindi facendo un MOV R15, R14 ossia **MOV PC, LR**

- 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).

Datapath istruzioni di branching

Nelle istruzioni B e BL si deve calcolare l'indirizzo a cui saltare alla prossima istruzione, quindi si deve calcolare questo indirizzo a partire dal registro R15, che contiene PC+8 a cui deve essere sommata l'immediate a 24 bit imm24 e poi si deve scrivere il risultato nel PC.

L'immagine viene moltiplicato per 4 ed esteso a 32 bit con segno. Quindi il segnale che pilota l'extender **ImmSrc viene esteso a 2 bit, nella codifica 10 il segnale estende la costante a 24 bit per l'istruzione di branch.**

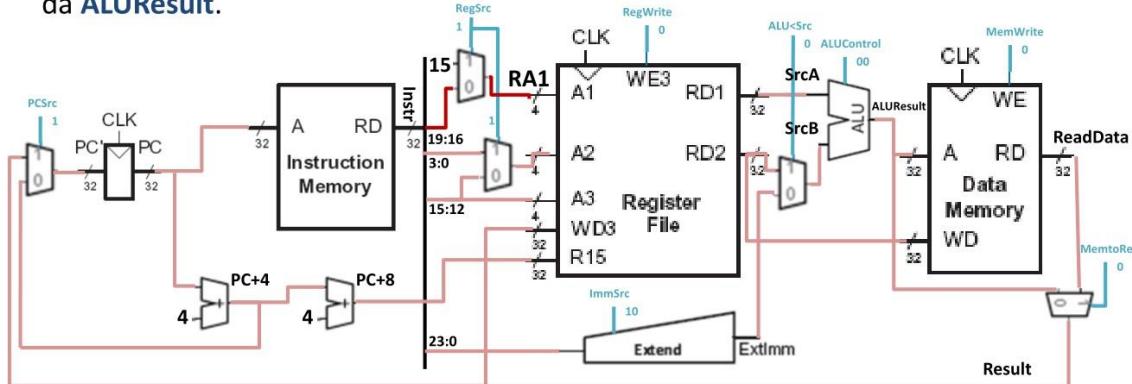
L'indirizzo PC+8 che è contenuto in R15 viene letto dalla prima porta del file register A1 e il valore viene dato poi in ingresso a RD1. Dunque un mux deve poter selezionare R15 come registro in ingresso alla porta A1, e non più Rn come di solito.

Dunque il mux è pilotato da un altro bit di RegSrc che seleziona i bit Rn Instr19:16 per le altre istruzioni e 15 per l'istruzione B, perchè è il registro che contiene PC+8 dato in input che deve essere sommato all'alu nelle istruzioni di branching. il risultato ottenuto deve essere inviato al pc che viene modificato

Dopo aver sommato l'indirizzo pc+8 all'immagine l'alu (alucontrol=00 perchè somma) produce il risultato ALUResult che passa al mux, il cui segnale **MemtoReg vale 0 perchè il risultato dell'alu viene riportato direttamente al program counter, che ha come segnale **PCSrc=1** dato che l'indirizzo della prossima istruzione da eseguire proviene direttamente dall'ALU**

RegWrite e MemWrite vengono settati a 0 perchè non c'è un'operazione di scrittura in memoria

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



Tutti i segnali di controllo del datapath:

- PCSrc
- MemtoReg
- MemWrite
- ALUControl
- ALUSrc
- ImmSrc
- RegWrite

Control Unit

La control unit è un'unità di controllo che genera i segnali di controllo del datapath, memorizza e aggiorna opportunamente le flag di stato.

E' costituita da una logica di combinatoria, nell'architettura a ciclo singolo. Nell'architettura a ciclo multiplo è un automa di mealy.

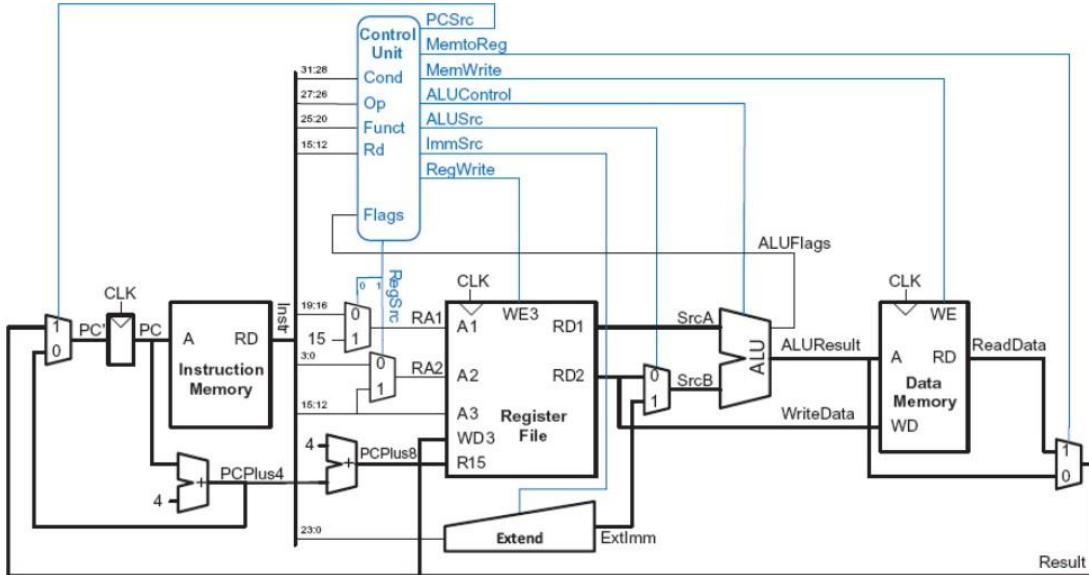
La CU genera i segnali di controllo prendendo in input:

- il campo **cond** dell'istruzione: gli serve per verificare se l'istruzione corrente deve essere verificata o meno sulla base dei flags
- il campo **op** dell'istruzione: che gli indica il tipo di istruzione da eseguire e in base ad essa la CU cambia i selettori
- il campo **funct** dell'istruzione
- il registro di destinazione **Rd**: per verificare se esso sia o meno R15(PC)
- i **flags** direttamente dall'ALU (**ALUFlags**): perchè la condizione viene verificata sulla base dei valori correnti dei flags di stato

La CU deve memorizzare anche i flag di stato attuali nel Current Program Status Register e li deve aggiornare in modo appropriato.

In output genera tutti i segnali di controllo che vanno a finire sul datapath e che consentono di eseguire le istruzioni di diverso tipo.

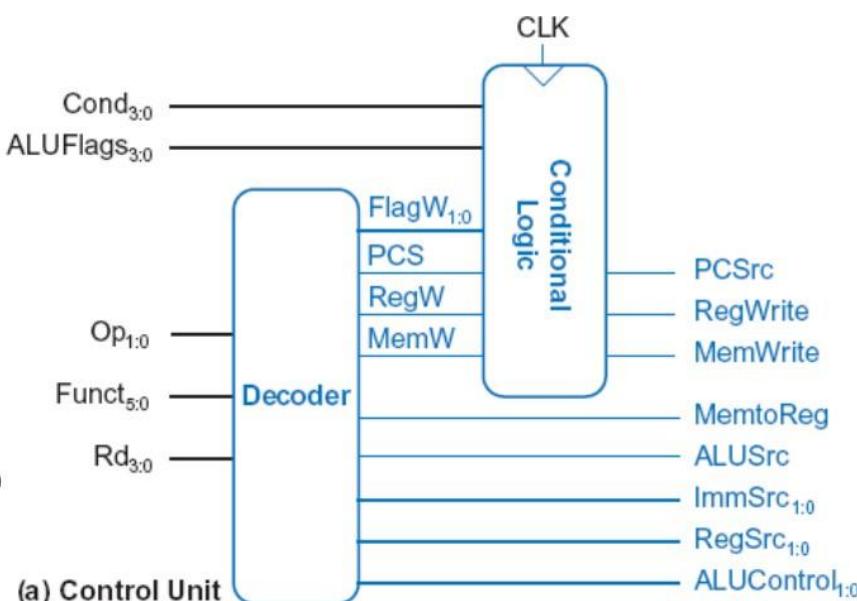
Schema completo datapath



Com'è fatta la CU all'interno

L'unità di controllo è divisa in 2 unità principali:

- **decoder**: che decodifica i valori di **op**, **funct**, **Rd** e genera i segnali di controllo
- **logica condizionale**: prende in input i **bit di cond** dell'istruzione e i **flag** che vengono dall'alu e verifica se rispetto ai valori memorizzati nel cpsr la condizione è soddisfatta o meno e poi **aggiorna i valori dei flag nel cpsr** nel caso in cui l'istruzione è tipo CMP, ADDS ecc. Quindi gestisce i flag di stato e li aggiorna quando l'istruzione deve essere eseguita su condizione.



Decoder della CU

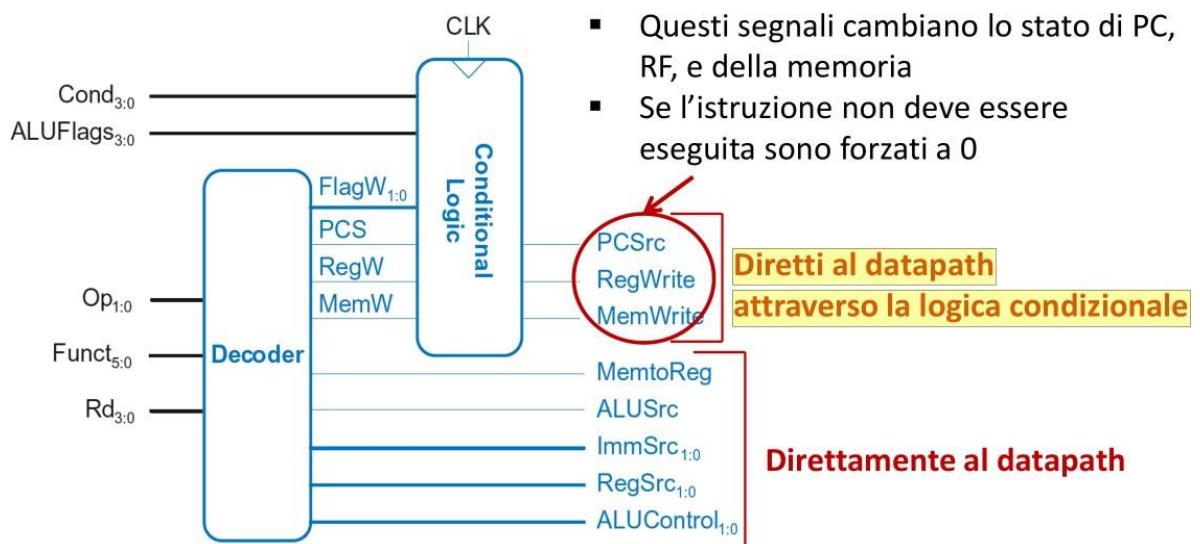
Dal decoder vengono inviati in output i vari segnali che vanno **direttamente sul datapath**, i quali sono *MemtoReg ALUSrc ImmSrc1:0 RegSrc1:0 ALUControl1:0*

Poi abbiamo altri output ossia FlagW1:0, PCS, RegW e MemW.

PCS, RegW e MemW diventeranno i veri output PCSrc, RegWrite e MemWrite solo se la condizione definita dai bit di cond e dagli aluflags e gestita dalla logica condizionale viene soddisfatta. Se la condizione non è soddisfatta questi segnali vengono sempre azzerati e quindi saranno uguali a 0 in output. **Diretti al datapath attraverso la logica condizionale**

Il Decoder della CU è inoltre 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 e quindi l'operazione che l'alu deve eseguire ;
- **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.



I 2 bit FlagW(1:0)

Sono i bit di FlagWrite Signal che indicano quando le ALUFlags devono essere aggiornate, consentendo anche l'aggiornamento del CPSR con i flag di stato, ovviamente quando il valore del bit S delle istruzioni di data processing è = 1.

Quindi quando S=1 le ALUFlags devono essere aggiornate.

Sono 2 bit perchè sappiamo che le operazioni aritmetiche come ADD,SUB aggiornano tutti i flags NZCV, mentre le operazioni logiche dell'ALU come AND e ORR non hanno necessità di aggiornare i bit C, V di Carry e Overflow, ma aggiornano solo i bit N e Z

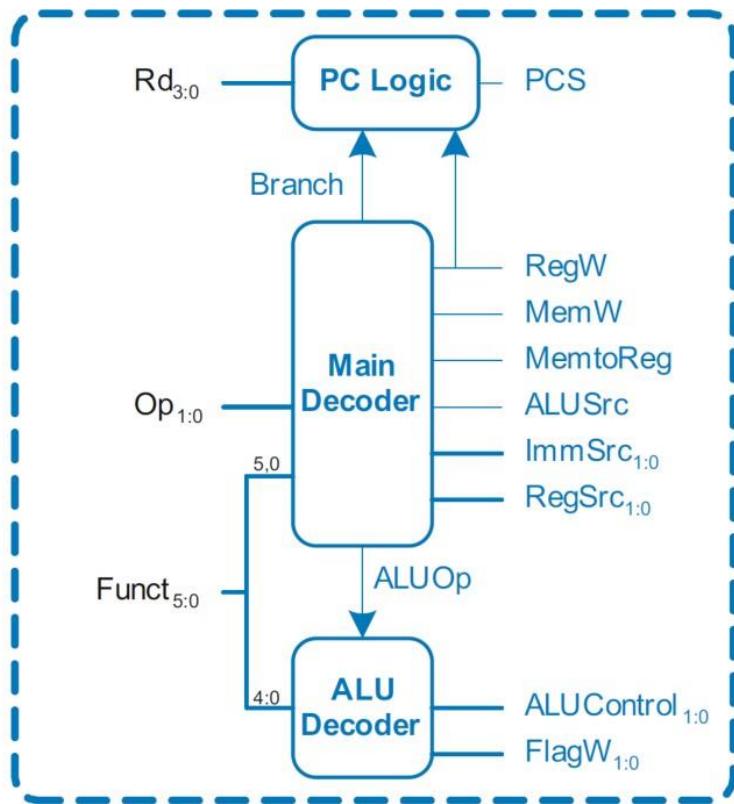
Quindi sono necessari due bit

- FlagW1 = 1: NZ (ALUFlags3:2 saved)
- FlagW0 = 1: CV (ALUFlags1:0 saved)

Es. istruzione ADDS, aggiorna tutti i flags => FlagW1=1 FlagW0=0
istruzione ANDS, aggiorna solo i flag NZ => FlagW1=1 FlagW0=0

I compiti del decoder ALU

- Main Decoder
- ALU Decoder
- PC Logic



Il decoder ha i seguenti compiti:

- **determinare il tipo di istruzione e se il secondo operando è un registro o un immediate:** 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, ImmSrc $_{1:0}$, e RegSrc $_{1:0}$.
- **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.

Il **Main Decoder** genera in particolare i valori ALUOp e Branch che indicano il tipo di istruzione, se è di branch o data processing. Ha in ingresso Op e Funct

L' **ALU Decoder** indica il tipo di operazione da eseguire (ADD, SUB ..) e genera il segnale ALUControl e i FlagW che indicano i flag da aggiornare

Il **PCLogic** genera il segnale PCS che deve passare per la logica condizionale prima di diventare PCSource (indica se il valore successivo del PC deve essere l'istruzione successiva PC+4 oppure se proviene dall'ALU nel caso di branch perché fa saltare la prossima istruzione in un punto diverso). Ha in ingresso Rd, e si attiva se Rd=15 (registro R15=PC).

La PC logic controlla se l'istruzione comporta una scrittura in R15 (quindi devo aggiornare anche PC) e RegW deve essere settato a 1 perché dobbiamo scrivere nel register file oppure è un branch secondo la condizione

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

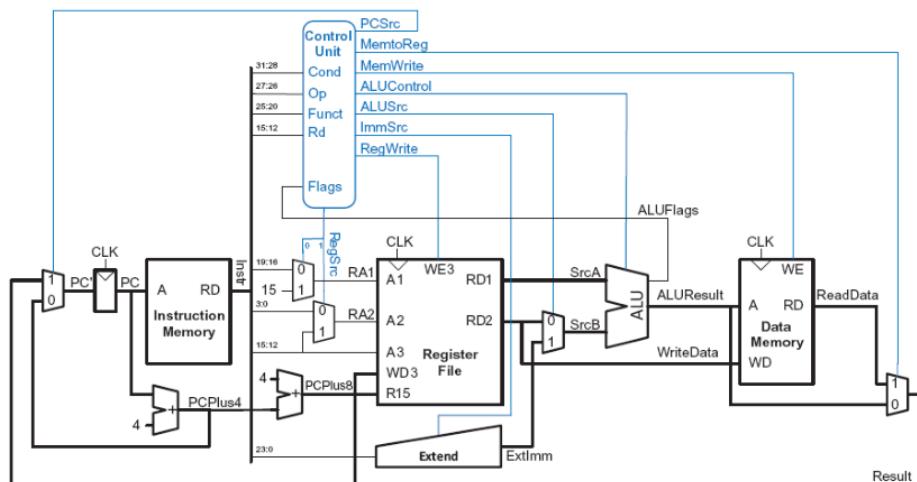
In uscita abbiamo PCS

$\text{PCS}=0 \Rightarrow \text{PCSrc}=0 \Rightarrow$ il PC si aggiorna in modo standard a $\text{PC}+4$

$\text{PCS}=1 \Rightarrow \text{PCSrc}=\text{PCS} \Rightarrow$ l'istruzione è eseguita e dobbiamo eseguire un branch

Importante

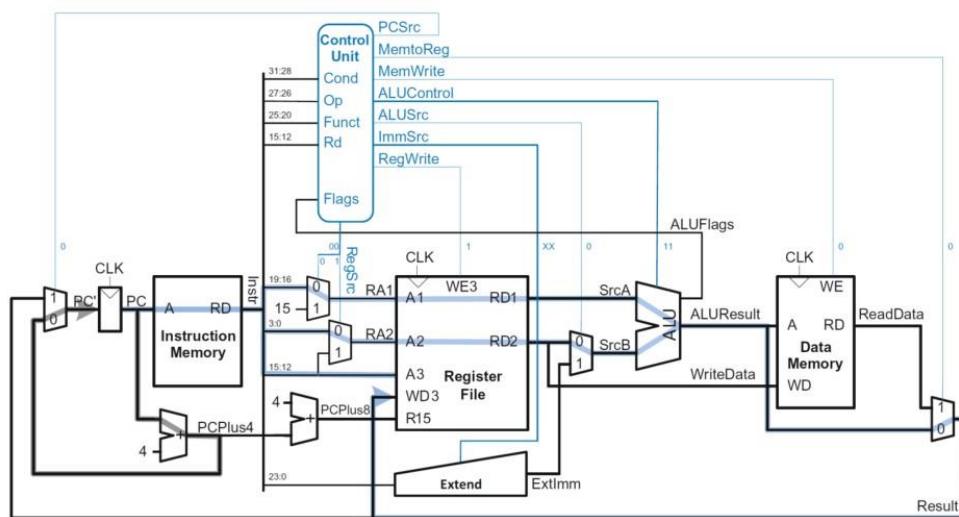
Op	Funct ₅	Funct ₀	Type	Branch	MemoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
11	X	X	B	1	0	0	1	10	0	X1	0



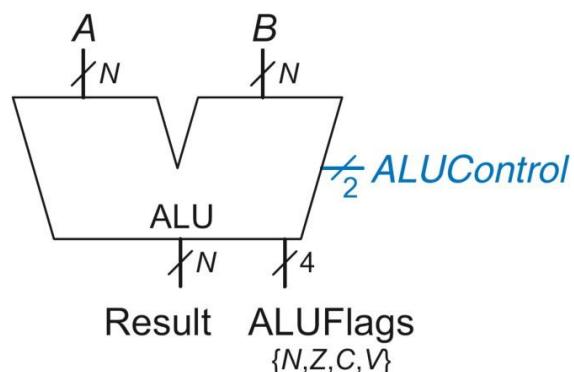
Dataflow

Esempio: DReg

Op	Funct ₅	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp	
00	0	X	DP Reg	0	0	0	0	XX	1	00	1



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

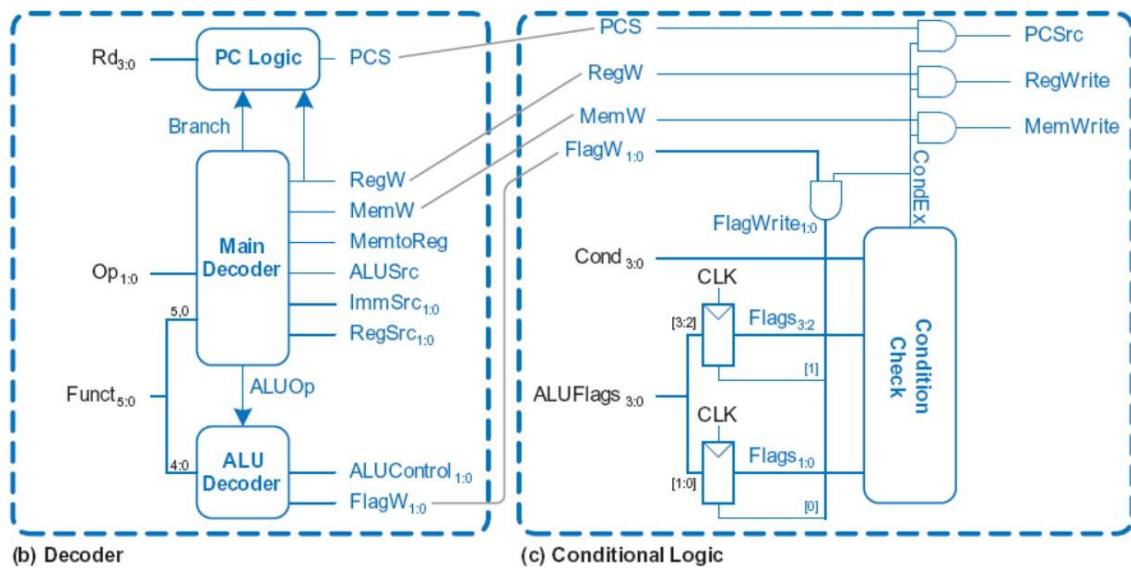
- $\text{FlagW}_1 = 1$: NZ ($\text{Flags}_{3:2}$) devono essere salvate
- $\text{FlagW}_0 = 1$: CV ($\text{Flags}_{1:0}$) devono essere salvate

Logica condizionale della CU

La logica condizionale è quella parte dell'unità di controllo che si occupa di verificare che la condizione codificata nei 4 bit più significativi di un'istruzione (cond) sia effettivamente soddisfatta e che in tal caso abilita la scrittura e l'aggiornamento dei flag rendendoli operativi e con i rispettivi segnali sul datapath.

Quindi determina se l'istruzione deve essere eseguita in base al campo cond dell'istruzione e ai valori attuali delle flag NZCV (i bit flags3:0). Se l'istruzione non deve essere eseguita le abilitazioni alla scrittura e il segnale PCSrc sono forzati a 0. La logica condizionale aggiorna anche alcune o tutte le flag ai valori ALUFlags quando FlagW è attivato dal decoder dell'alu e la condizione di esecuzione dell'istruzione si è verificata.

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

Le condizioni sono stabilite partendo dall'istruzione CMP che fa la sottrazione del primo argomento col secondo argomento. Per ogni mnemonico di condizione è associata un'espressione booleana nei flags di stato NZCV

Dopo aver eseguito l'istruzione CMP i flags vengono aggiornati nel CPSR che serve alla logica condizionale per prelevare i valori NZCV per calcolare le espressioni delle condizioni ciò avviene anche aggiungendo il suffisso -S all'istruzione, solo che il risultato dell'istruzione poi viene salvato in un registro

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

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	$Z \text{ OR } \bar{C}$
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	$Z \text{ OR } (N \oplus V)$

La differenza A-B si fa col CMP

Dimostrazione delle CondEX degli mnemonici

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

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	$Z OR (N \oplus V)$

Per dimostrare questi mnemonici si parte dall'operatore **LT** (signed less than)

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

Questa condizione può generare overflow oppure no

-Quando $A - B$ non genera overflow allora il risultato $A - B$ è deve essere negativo per avere che $A < B$. Quindi faccio $A - B$, verifico che non ci sia overflow (bit $V = 0$ dei flags) e se il risultato è $<$ di 0 (ossia il bit $N = 1$) allora la condizione è rispettata. Questo accade sempre quando A e B hanno lo stesso segno. **Quindi se $V=0$, deve esserci $N=1$**

-Può accadere che $A - B$ generi overflow: in tal caso avremo che $V=1$. Ciò significa che A e $(-B)$ hanno segni diversi. In questo caso $A < B$ è vera sse A negativo e B positivo, cioè A negativo e $(-B)$ negativo. Essendoci overflow il risultato sarà per forza positivo, discorde dai due segni **quindi se $V=1$, deve esserci $N=0$.**

Ecco che abbiamo N xor V

Ricaviamo ora **LE (signed less than or equal)**

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

Un operando A è minore uguale di un altro B se e solo se **$A - B < 0$** ossia è rispettata la condizione di **LT (N xor V) oppure (or, +) se $A - B == 0$** cioè A e B sono uguali, in questo caso si aggiunge il bit di flag $Z=1$.

Le altre due condizioni **GE** e **GT** si ottengono per negazione delle precedenti

Dimostrazione dei mnemonici che valgono per interi senza segno

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	$Z \text{ OR } \bar{C}$

Problema per l'istruzione CMP, 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 . Questo vale per numeri con segno.

-Nella rappresentazione senza segno invece per fare la sottrazione, complementare bit a bit corrisponde a fare **1-bit corrente, quindi 2^n-1-B**

$$\sim(X_{N-1}\dots X_0) = (1-X_{N-1})\dots(1-X_0) = 1\dots 1 - (X_{N-1}\dots X_0) = 2^N - 1 - B$$

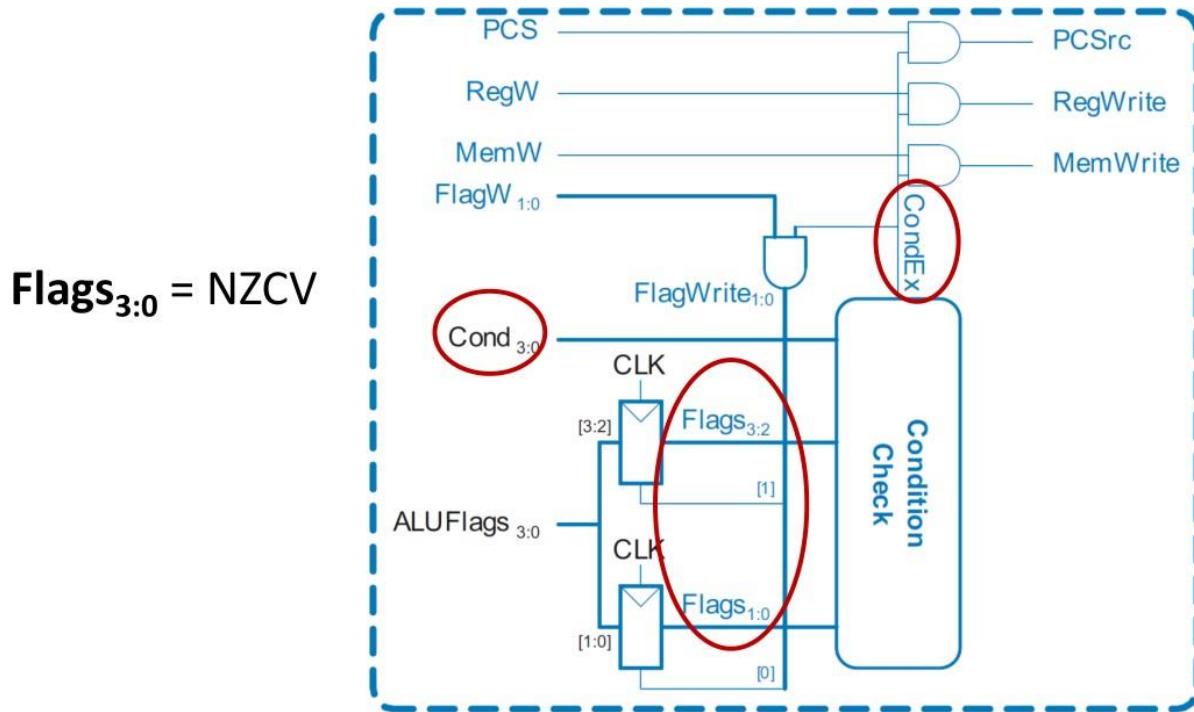
- Quindi $A+(\sim B+1)$ corrisponde nella rappresentazione senza segno a 2^N+A-B

Perchè **-1 e +1 si semplificano**

- Ora 2^N+A-B non genera un carry out sse $2^N+A-B \leq 2^N-1$ sse $A+1 \leq B$ sse $A < B$
- Quindi $A < B$ è verificato dalla condizione \bar{C}
- $A \leq B$ sse $A < B$ o $A = B$ sse $Z + \bar{C}$
- $A > B$ sse $!(A \leq B)$ sse $\bar{Z}C$

$$A \geq B = !(A < B) \Rightarrow \text{NOT}(\text{NOT } C) = C$$

Logica condizionale dal punto di vista architettonico



Prende dal decoder i valori in input PCS, RegW, MemW, FlagW_{1:0} , i 4 bit di Cond e ALUFlags. Ha come output PCSrc, RegWrite, MemWrite.

E' composta da una logica che verifica la condizione "condition check" che prende in input i bit di cond dell'istruzione perchè per ogni codice diverso verificheremo una condizione diversa su flag di stato. Poi prende i 4 flags ALUFlags del CPSR che sono CV (3:2) e NZ (1:0). in base ai flag verifica se la condizione è soddisfatta o meno.

Es. Se abbiamo il codice MI prende il valore dei flag NZ, in particolare valuta N e se esso è uguale a 1 il valore di uscita CondEx sarà uguale a 1, altrimenti condex=0.

Il valore di uscita del condition check Condex va in and con tutte e tre le uscite di output PCSrc, RegWrite e MemWrite e quindi se vale 0 gli output vengono forzati a 0 e non dovremo aggiornare i flag di stato, quindi nessun registro o memoria.

Se il valore di uscita Condex=1 e quindi la condizione è rispettata i segnali PCS RegW e MemW diventano operativi e quindi si tramutano nei vari segnali finali di output uguali 1 e diventano operativi, essendo riportati nel datapath. Nel caso S=1 nelle istruzioni di data processing si aggiornano anche i flags nel CPSR

I flagw valgono 1 nel caso di operazioni aritmetiche (aggiornano tutti i flags). Valgono 10 per operazioni logiche che fanno sì che si aggiornano solo NZ. In questo caso l'abilitatore flags1:0 va a 0

Analisi delle prestazioni

Ogni istruzione nel processore a ciclo singolo impiega un ciclo di clock, quindi il **CPI è 1**. Ogni porzione del datapath e la control unit producono dei ritardi.

Ad esempio c'è un ritardo dovuto al caricamento del nuovo indirizzo nel PC nel fronte di salita del clock.

C'è un ritardo di lettura dell'istruzione in memoria, dovuto al decoder che deve impostare i segnali e gli abilitatori delle memorie e dei vari mux in modo opportuno per far sì che il datapath dell'istruzione sia corretto.

Inoltre lungo il datapath ci sono 2 ritardi che avvengono in contemporanea ad esempio quando abbiamo un'istruzione di tipo LDR da una parte dobbiamo ricavare il base address da registro e dall'altra l'offset dell'immediate. Queste istruzioni sono eseguite in parallelo e producono ritardi dovuti al mux, al tempo di risposta in lettura dal file register, ritardo dovuto all'extender per l'estensione dell'immediate a 32 bit.

Abbiamo un ritardo dovuto all'ALU che deve eseguire l'operazione. Ritardo in scrittura sul register file.

Tutti i ritardi tra loro quando si verificano costituiscono diversi **critical path per l'istruzione LDR**

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.

Sommendo tutti questi tempi parziali otteniamo il tempo di ritardo totale che il processore a ciclo singolo impiega per eseguire l'istruzione di LDR

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

In realtà se analizziamo quanto valgono i ritardi notiamo che il ritardo in lettura dal register file Trfread è molto maggiore del ritardo che impiega l'extender ad estendere una parola. questo vuol dire che nella maggior parte delle implementazioni l'alu, la memoria ed il register file sono sostanzialmente più lenti di tutti gli altri blocchi combinatori, per tanto 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

2 perchè facciamo 2 accessi in memoria
e passiamo in 2 mux

Limiti architetture a ciclo singolo

Le architetture a ciclo singolo hanno 3 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 architetture a 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 processore MULTICICLO

considereremo un limitato set di istruzioni:

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

LDR

Si riuniscono istruzioni e dati in un'unica memoria.

Si suddivide l'istruzione di LOAD in più fasi.

La prima fase è quella di FETCH.

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

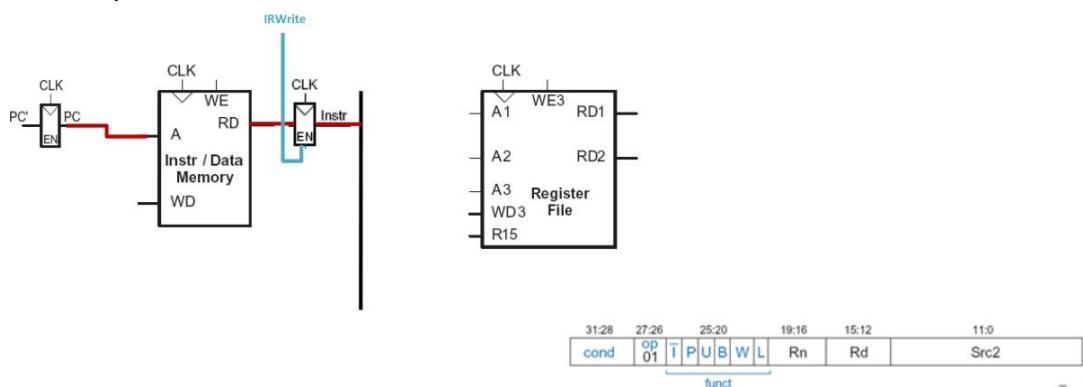
Esso è associato alla memoria dati/istruzioni, quindi il suo output PC è l'input della porta A della memoria, che riceve l'indirizzo dell'istruzione da eseguire.

La memoria dato l'indirizzo A ritorna in output il contenuto sulla porta RD1, cioè l'istruzione.

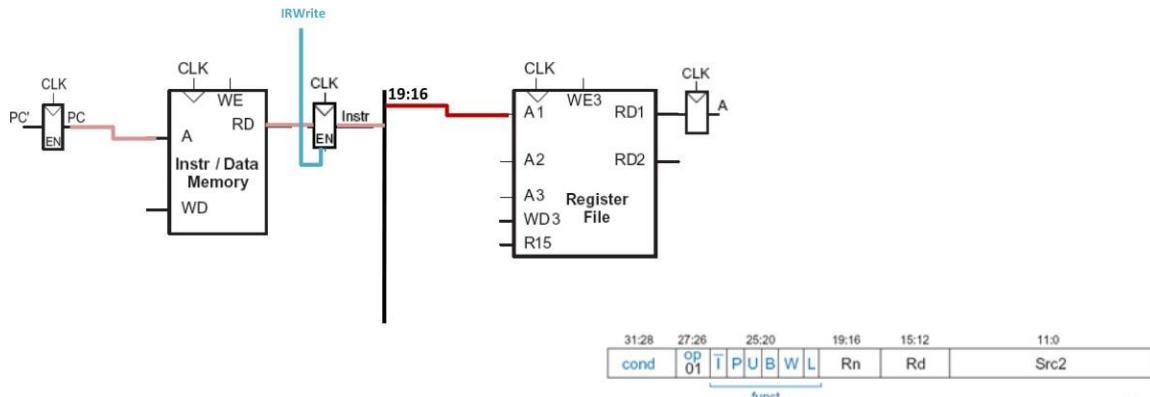
La porta RD1 è collegata ad un registro IR (Instruction Register).

L'istruzione a 32 bit viene letta e memorizzata nel registro IR.

Il registro IR riceve un segnale IRWrite che indica quando caricare una istruzione. In questo caso, nella fase di fetch, è uguale a 1. Mettiamo un registro perché vogliamo suddividere le varie fasi in più clock.



Il passo successivo è quello di leggere il registro sorgente contenente l'indirizzo di base. Questo registro è specificato nel campo Rn dell'istruzione, Instr19:16. I 4 bit Rn vengono collegati all'ingresso della porta indirizzo del file register, (A1). La porta A1 del file register riporta in output RD1 il valore del base address. Il valore del base address viene memorizzato in un registro A dal file register.

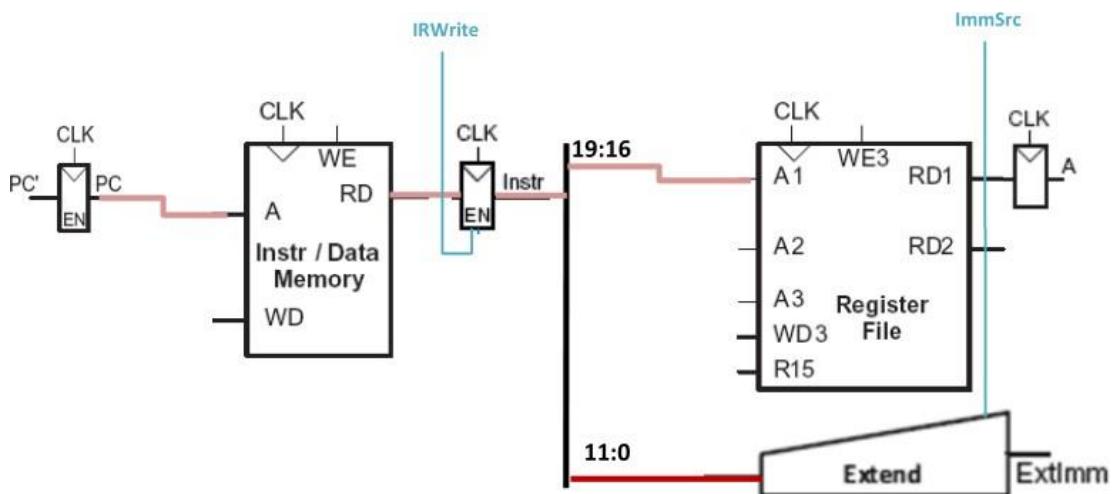


Contemporaneamente, sappiamo che l'istruzione LDR richiede anche un offset, il quale è memorizzato nell'istruzione stessa e corrisponde ai bit Instr11:0, ossia l'Immediate a 12 bit. Si prendono i 12 bit di immediate dell'istruzione e si estendono a 32 bit (unsigned) tramite la logica Extend per ottenere l'offset.

Il valore a 32 bit (ExtImm) è tale che ExtImm31:12 = 0 e ExtImm11:0 = Instr11:0.

ExtImm estende a 32 bit costanti a 8, 12 e 24 bit, quindi usiamo il selettore ImmSrc per scegliere la tipologia di offset da estendere, per istruzioni di memoria l'offset da estendere è a 12 bit unsigned.

Il valore esteso ExtImm non viene memorizzato in un registro, poiché dipende solo da Instr, che non cambia durante l'esecuzione dell'istruzione.

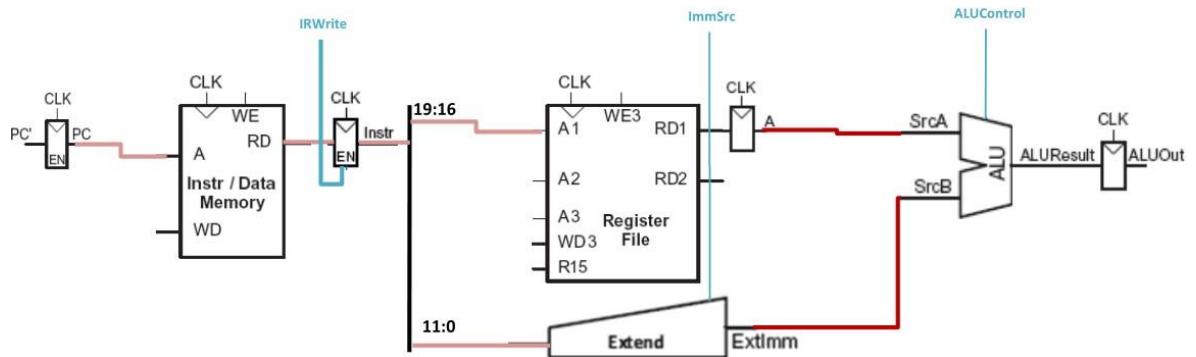


Nella fase successiva il processore deve sommare l'indirizzo di base all'offset per trovare l'indirizzo di memoria a cui leggere il dato da caricare nel registro di destinazione Rd.

La somma è effettuata dall'alu.

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

L'alu genera un risultato a 32 bit ALUResult ossia l'indirizzo di memoria da cui fare il load del dato, il quale viene salvato in un registro ALUOut



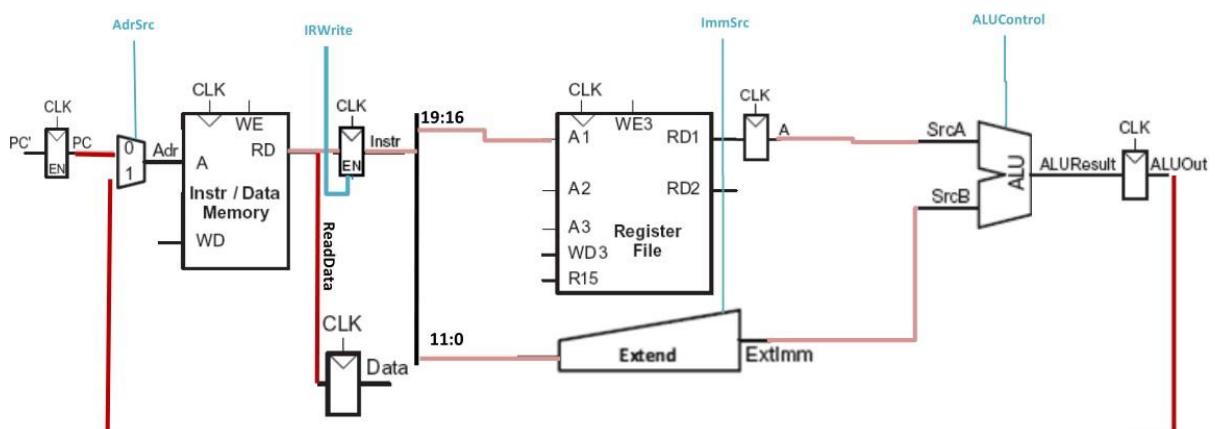
A questo punto si prende l'indirizzo generato dall'ALU e contenuto nel registro ALUOut e lo si invia alla porta A della memoria, che deve perlevare il dato a quell'indirizzo. In questo caso poiché la memoria si utilizza sia per caricare l'istruzione che per andare a prendere il dato, abbiamo bisogno di un mux che disambigua l'accesso alla memoria con ALUOut o PC. Il multiplexer è controllato dal segnale AdrSrc.

Nella fase di fetch il mux sarà 0 => viene caricata l'istruzione in memoria tramite il pc

Nella fase di caricamento del dato il mux sarà 1 poiché deve essere caricato in memoria l'indirizzo ALUOut.

Nel ciclo multiplo gli abilitatori assumono diversi valori in fasi diverse, non sono sempre stabili ad un valore.

Il contenuto del dato viene letto dalla memoria dati e inviato sul bus ReadData, e poi viene memorizzato in un registro chiamato Data.



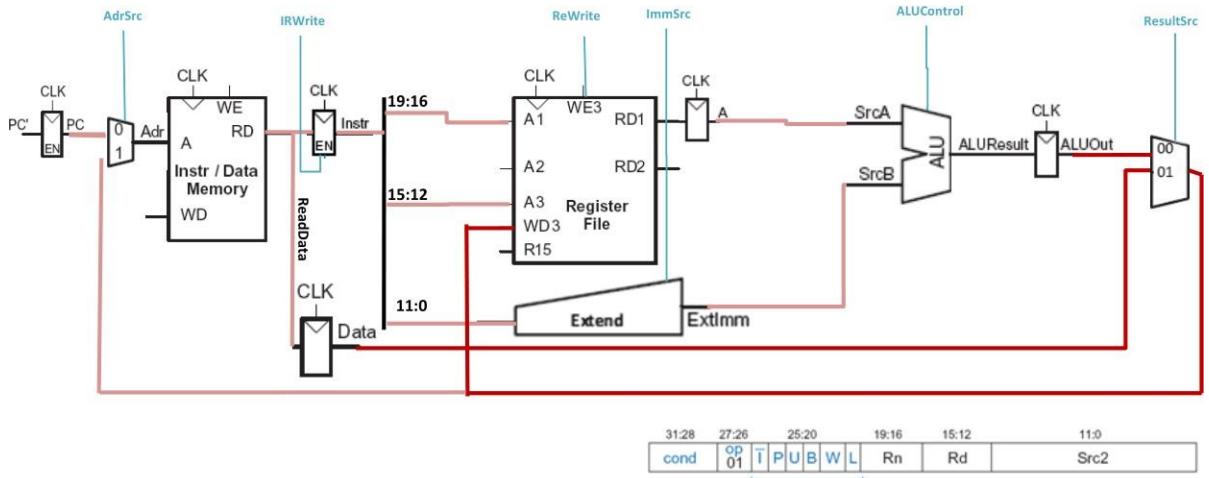
Ora il dato deve essere scritto e salvato nel file register nel registro indicato all'indirizzo contenuto nel campo Rd dell'istruzione, che è il registro di destinazione. Il registro Rd in cui scrivere tale dato è specificato dai bit 15:12 dell'istruzione Instr.

Piuttosto che collegare direttamente il Data alla porta di scrittura WD3 del file register , facciamo passare Data in un mux perchè WD3 può ricevere sia in ingresso dal registro Data ma anche direttamente dall'ALU, come per l'istruzione di data processing ADD . Quindi aggiungiamo un mux che seleziona fra ALUOut e Data.

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

Se il mux ha il segnale ResultSrc = 00 collega ALUOut a WD3.

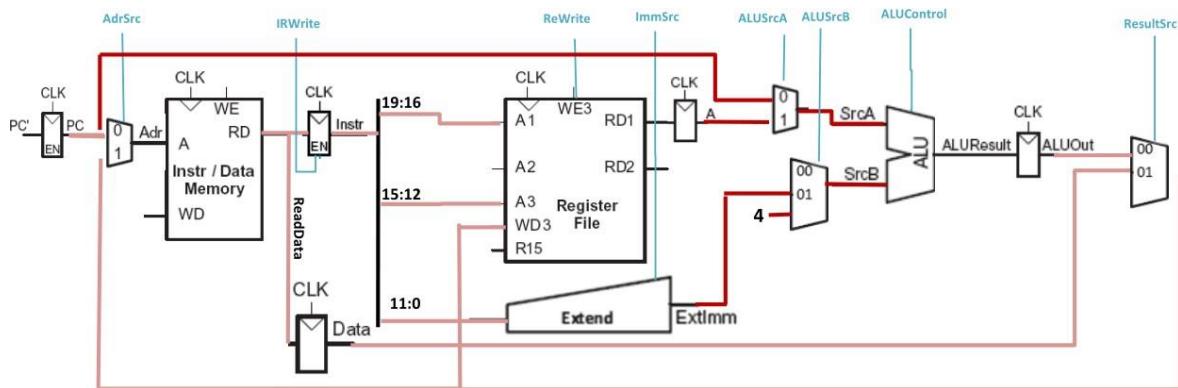
Se il mux ha il segnale ResultSrc = 01 collega Data a WD3



Un ulteriore compito a carico dell'ALU è l'incremento del PC, operazione che prima era svolta da una ALU diversa. Il valore del PC deve essere riportato verso l'alu aggiungendo un mux per sceglierlo in input il suo ingresso. Il contenuto del PC deve essere poi sommato a +4, che deve essere l'ingresso SrcB dell'alu per aggiornare il valore del PC.

Quindi aggiungiamo un mux sul primo ingresso dell'alu che permette di scegliere fra il contenuto del registro A e il PC. Sul secondo ingresso SrcB aggiungiamo un ulteriore multiplexer che permetta di selezionare fra ExtImm e la costante 4.

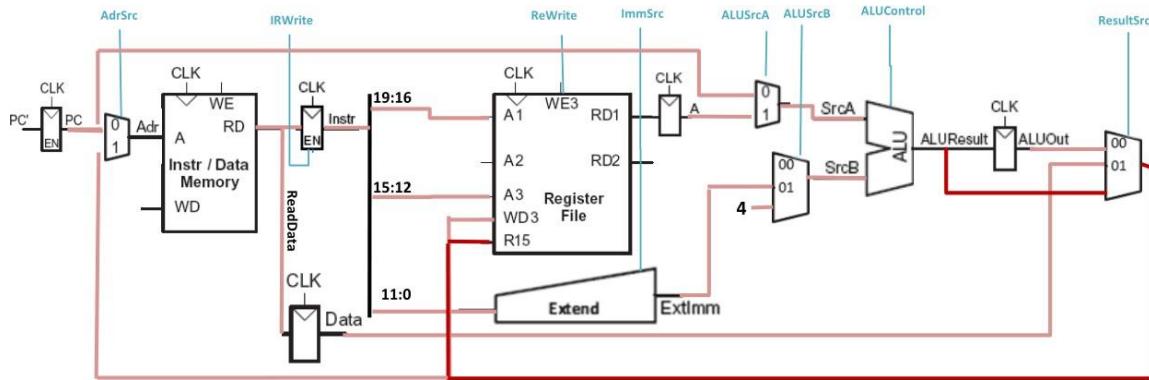
L'aggiornamento del PC avviene durante la fase di fetch e decode perché l'alu non è ancora impegnata per la somma base address+offset.



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

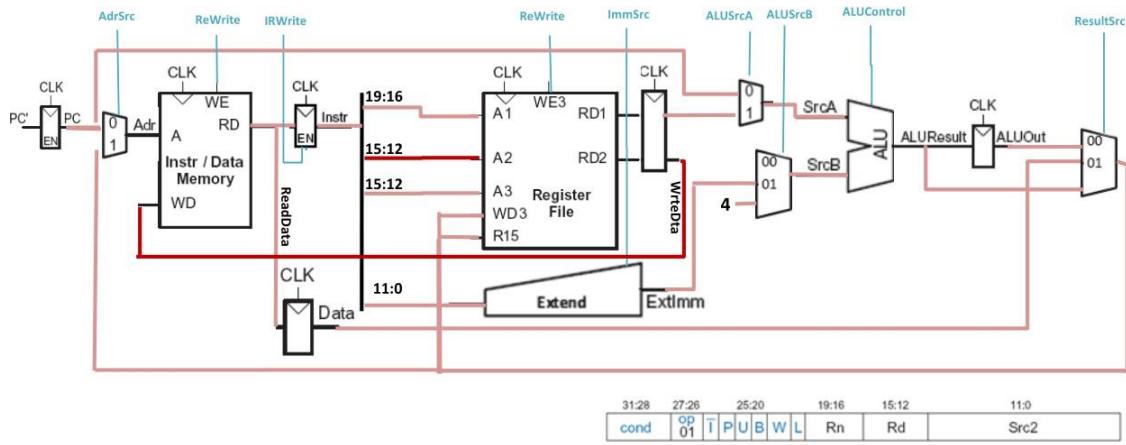
Analogamente all'istruzione di caricamento, STR legge l'indirizzo di base dalla porta RD1 del register file, estende la costante e l'ALU somma i due valori per calcolare l'indirizzo di memoria. Tutte queste operazioni sono già supportate.

In aggiunta, STR legge il registro Rd da cui scrivere, che è specificato nei bit Instr15:12.

I bit corrispondenti a Rd finiscono all'ingresso A2 del register file da cui si prende il contenuto da memorizzare in memoria sull'uscita RD2.

Il contenuto di RD2 è inserito in un registro temporaneo WriteData e al passo successivo è inviato alla memoria, con segnale MemWrite attivo, sull'ingresso WD.

Ciclo in meno rispetto a LDR



Datapath data processing

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

Per le istruzioni di data processing con registro (ADD, SUB, AND, OR), il datapath legge il secondo operando specificato da Rm, indicato nei bit Instr3:0.

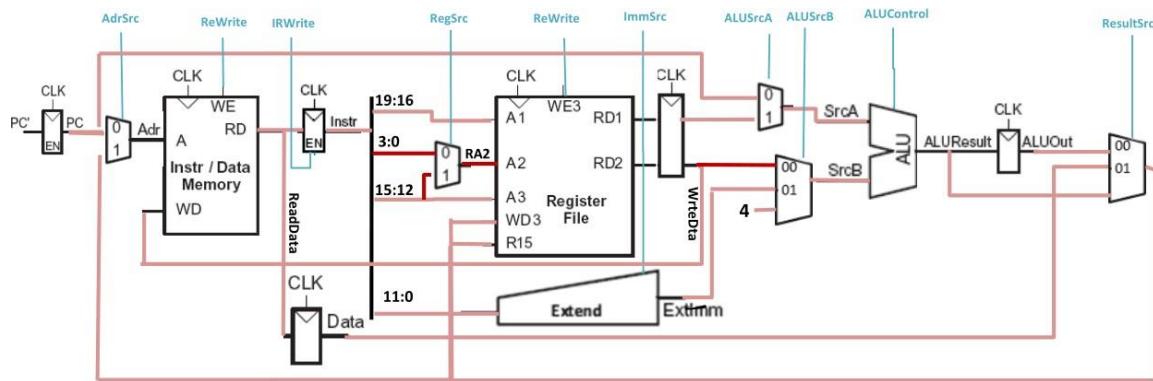
Inseriamo un multiplexer per selezionare tale campo sulla porta A2 del register file. Il mutiplexer è controllato dal segnale RegSrc. Inoltre, estendiamo il multiplexer in SrcB in modo da considerare questo caso.

ALUSrcB:

00 - SrcB da registro

01 Src da extender

4 - aggiorna pc

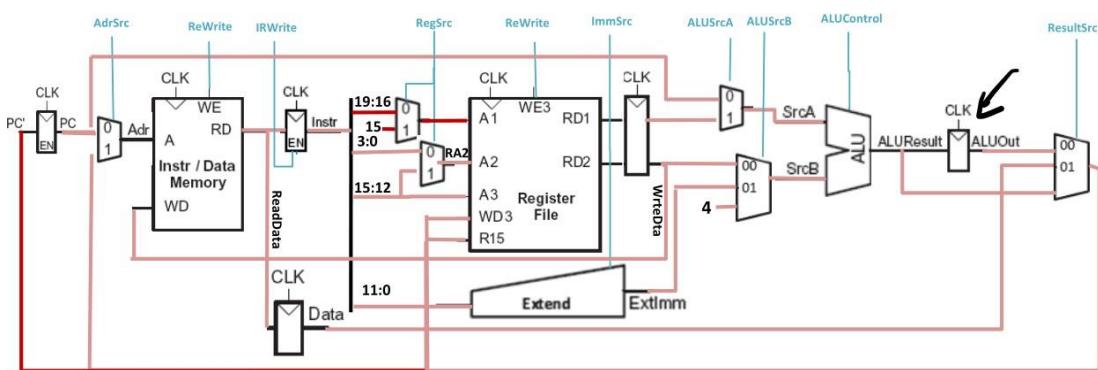


Datapath branch

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 ciclo multiplo

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 (Instr31:28, Instr27:26, e Instr25:20), 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.

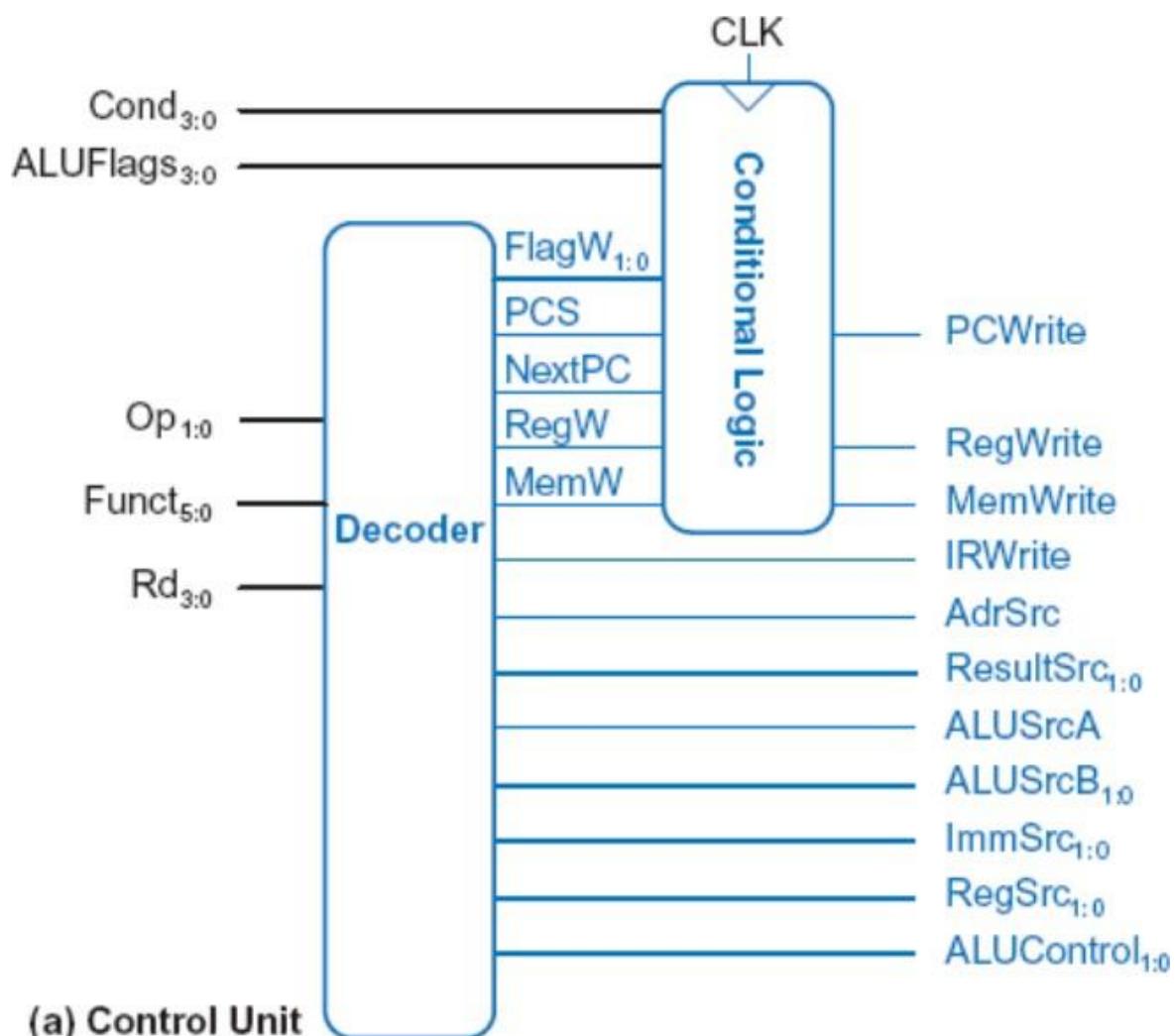
Come nel processore a ciclo singolo, l'unità di controllo è suddivisa in decodificatore e logica condizionale. Il decodificatore è progettato come una FSM, che produce i segnali appropriati per i diversi cicli, sulla base del proprio stato.

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

I valori dei selettori cambiano fase dopo fase , il decoder deve essere una macchina a stati finiti che stato per stato , fase per fase, aggiorna i valori di output dei segnali di controllo in modo differente

Il decoder principale viene quindi sostituito nel processore multiciclo da una FSM principale, che deve produrre la sequenza di segnali di controllo nei passi opportuni. Si progetta questa macchina di moore in modo che le sue uscite siano in funzione del solo stato presente.

Il decoder dell'alu e la logica del pc sono identiche a quelli del processore a ciclo singolo. La logica condizionale è quasi uguale a quella del processore a ciclo singolo: serve solo il segnale aggiuntivo NextPc per forzare una scrittura nel PC quando si calcola PC+4



Dataflow

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.

La FSM deve generare i segnali di selezione dei multiplexer, le abilitazioni dei registri e i segnali di scrittura in memoria del datapath

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.

I segnali di abilitazione RegW, MemW, IRWrite e NextPC sono elencati solo quando devono essere attivati, cioè portati a 1. Se non sono elencati si assume che valgono 0.

Dataflow LDR-STR

Il primo passo di ogni istruzione è il fetch dalla memoria dell'istruzione all'indirizzo presente nel PC, e l'incremento del PC per puntare all'istruzione successiva.

La FSM si porta in questo stato denominato Fetch al reset.

Per leggere dalla memoria l'istruzione, AdrSrc=0 in modo che l'indirizzo sia preso dal PC
IRWrite è attivato per salvare l'istruzione nel registro istruzioni IR

In contemporanea il PC deve essere incrementato di 4 per puntare all'istruzione successiva.

Il processore può adoperare l'ALU perchè inutilizzata, la usa in parallelo alla fase di fetch per calcolare PC+4.

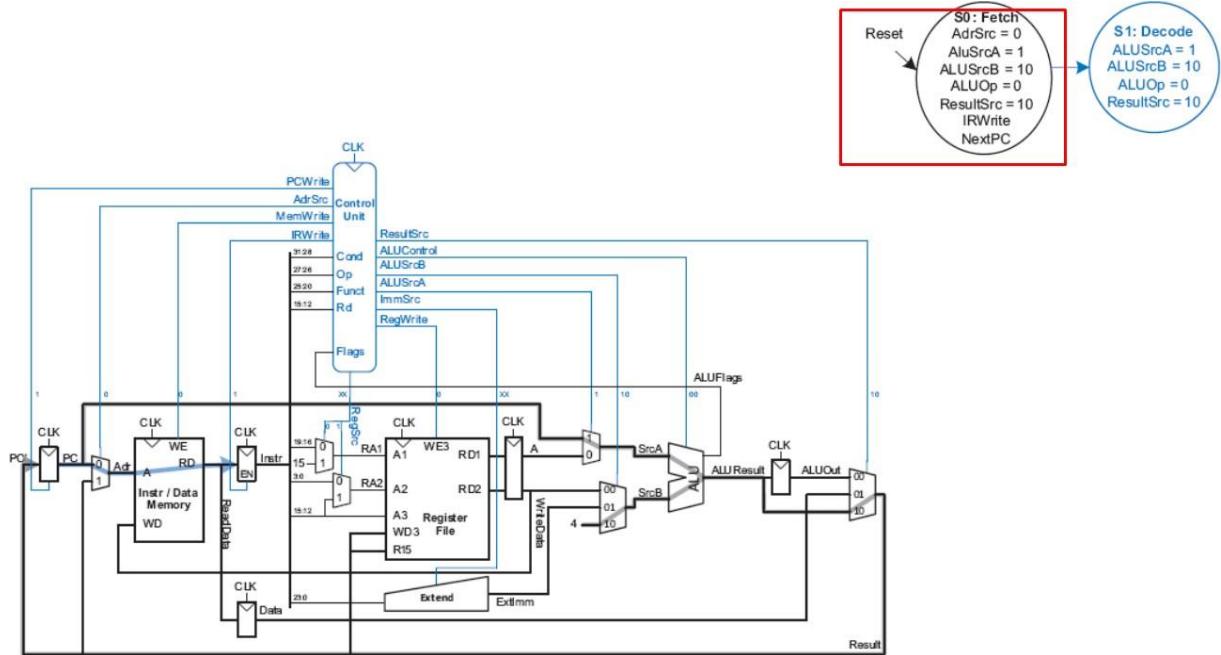
Gli ingressi dell'alu ALUSrcA=1 in modo che provenga dal PC l'indirizzo;
ALUSrcB=10, quindi si seleziona la costante 4 da sommare all'indirizzo del PC.

ALUOp=0, ALUControl=00 quindi si effettua la somma di ALUSrcA e ALUSrcB.

Per aggiornare il PC col nuovo valore PC+4, il segnale ResultSrc deve essere 10 per selezionare dirattamente il risultato dell'alu ALUResult.

Inoltre NextPC=1 perchè si deve abilitare la scrittura nel PC del nuovo indirizzo, PCWrite=1.

Quindi si aggiorna il PC e si effettua il fetch dell'istruzione.



Si passa ora alla fase di Decode.

Si deve leggere il secondo operando dal register file oppure come immediato e si deve decodificare l'istruzione.

I registri e l'immediate sono selezionati da RegSrc e ImmSrc, che sono generati dal Decoder dell'ALU opportunamente sulla base dei bit Instr dell'istruzione.

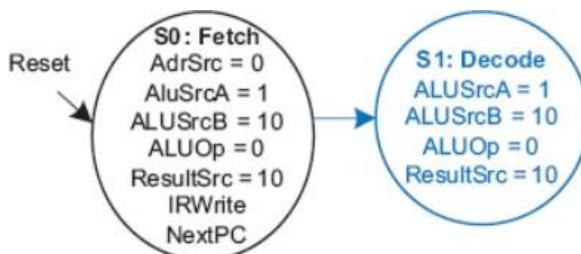
RegSrc0 vale 1 per i salti, per leggere 15 e quindi il registro R15 che contiene $Pc+8$ come SrcA da inviare all'alu. RegSrc1 è 1 per le istruzioni di scrittura in memoria STR, per leggere come SrcB il valore Rn da memorizzare

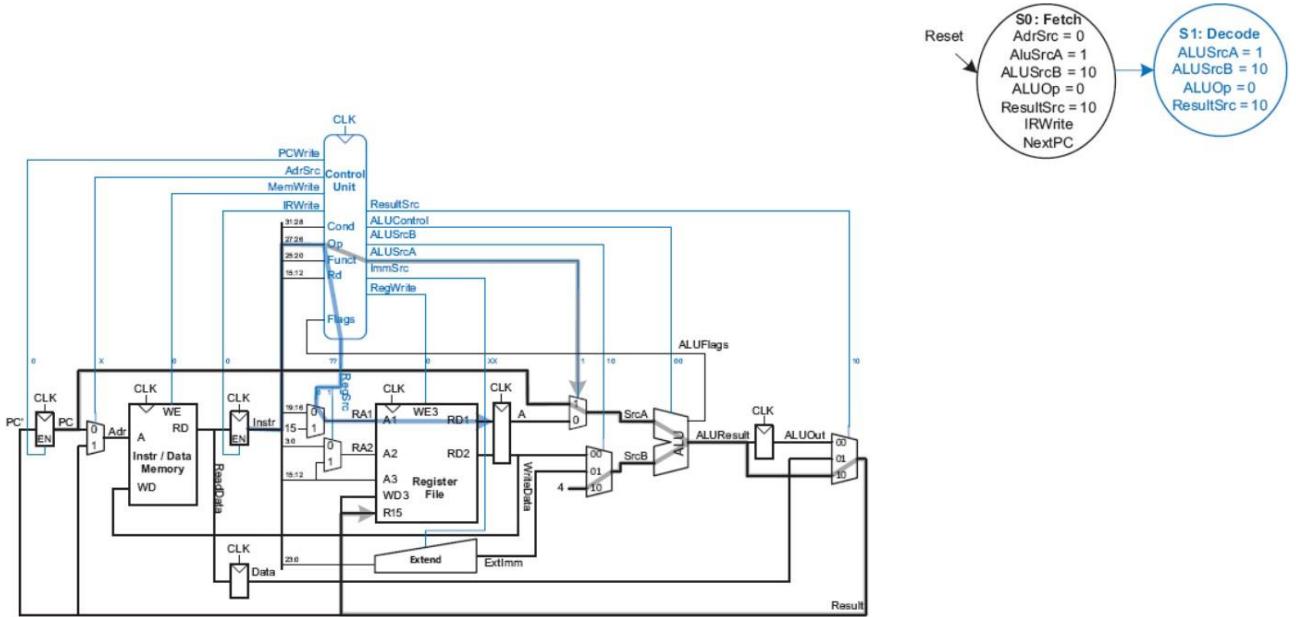
ImmSrc = 00 per istruzioni di data processing, 01 istruzioni di memoria, 10 salti

Nel mentre l'alu viene usata per calcolare il valore di R15 da aggiornare, $Pc+8$, sommando ancora 4 al PC già incrementato nello stato di Fetch.

I segnali di controllo sono generati in modo tale da selezionare il valore PC come primo ingresso dell'ALU ($ALUSrcA=1$) e 4 come secondo ingresso dell'alu ($ALUSrcB=10$) e viene attivata l'operazione di somma ($ALUOp=00$).

La somma viene selezionata direttamente dall'ALU e inviata all'ingresso di R15 del register file tramite Result, per cui $ResultSrc=10$.





Ora si passa allo stato di execute.

La FSM procede ad uno dei diversi possibili stati in base ai campi op e funct dell'istruzione esaminati durante il passo Decode.

Se l'istruzione è di memoria (STR o LDR, con op=01) il processore multiciclo deve calcolare l'indirizzo di Rd sommando all'indirizzo di base lo spiazzamento esteso dall'extender.

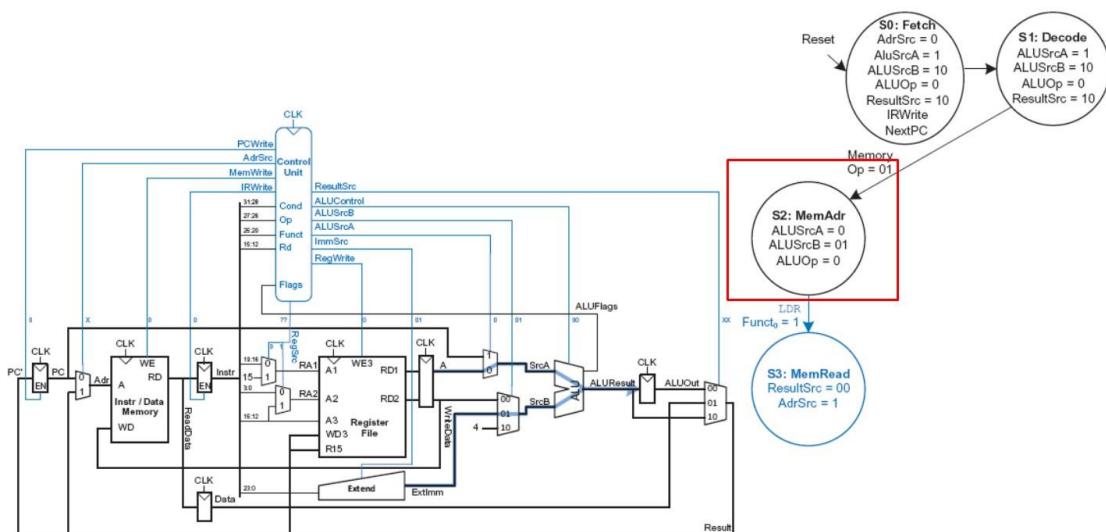
Questo richiede ALUSrcA=0 per selezionare l'indirizzo di base del regista, dal file register, che poi è memorizzato temporaneamente nel regista A.

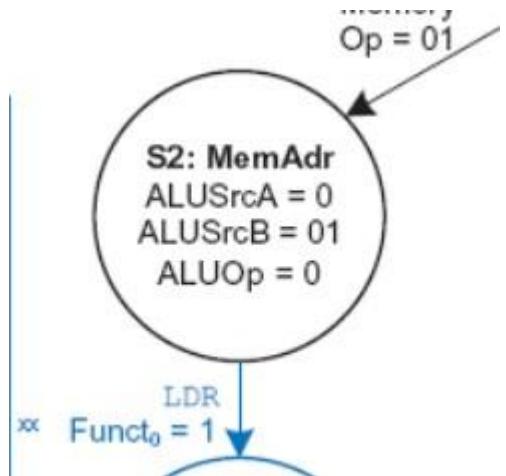
ALUSrcB=01 per selezionare l'immagine estesa da sommare al base address, ExtImm.

ALUControl=00 per effettuare la somma.

L'indirizzo calcolato viene memorizzato nel regista ALUOut per i passi successivi.

Operazione: Execute (memory address computation)





A questo punto la FSM ha due possibilità in base al tipo di istruzione:

Se l'istruzione è LDR (funct L=1) il processore multiciclo deve leggere un dato dalla memoria e scriverlo nel register file.

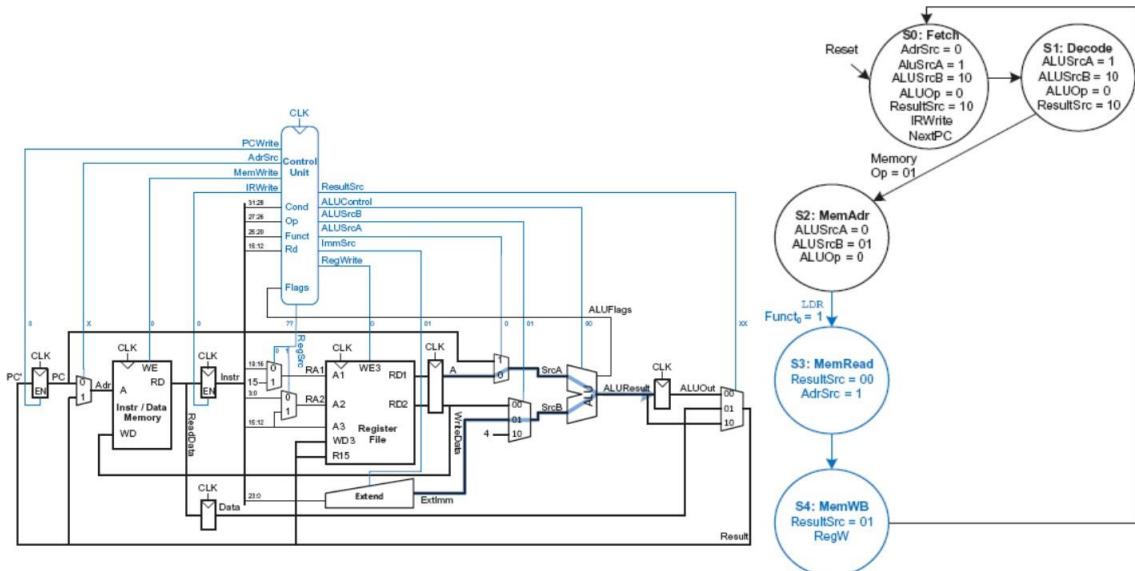
Per leggere dalla memoria ResultSrc=00 per selezionare il risultato dal registro ALUOut che contiene l'indirizzo da passare alla memoria da cui poi si va a leggere.

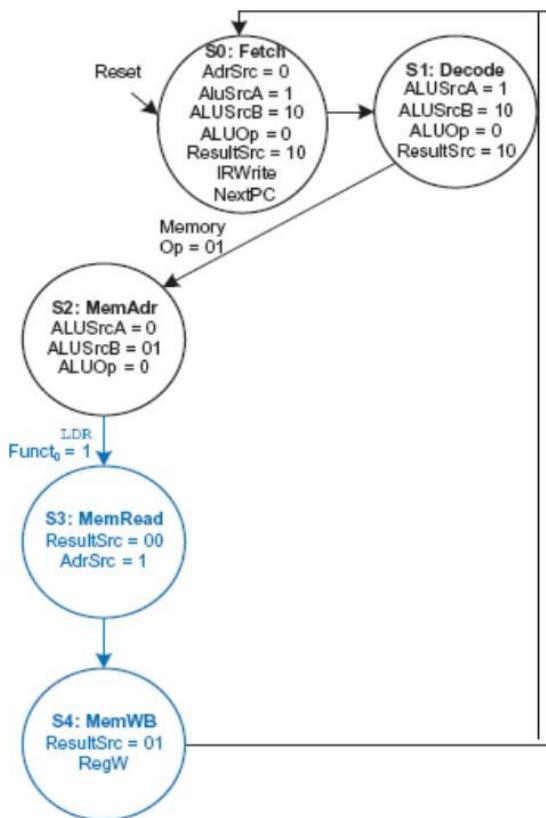
AdrSrc=1 per selezionare l'indirizzo di memoria Result appena calcolato e salvato in ALUOut. Il contenuto della parola indirizzata viene letto dalla memoria e salvato nel registro Data durante il passo MemRead.

Nel passo di scrittura finale MemWB , il contenuto di Data viene scritto nel register file all'indirizzo di Rd sulla porta WD3.

ResultSrc=01 per selezionare Result da Data e viene attivato RegW per scrivere nel register file, completando l'esecuzione dell'istruzione di LDR.

Infine la FSM torna allo stato di fetch per iniziare l'istruzione successiva



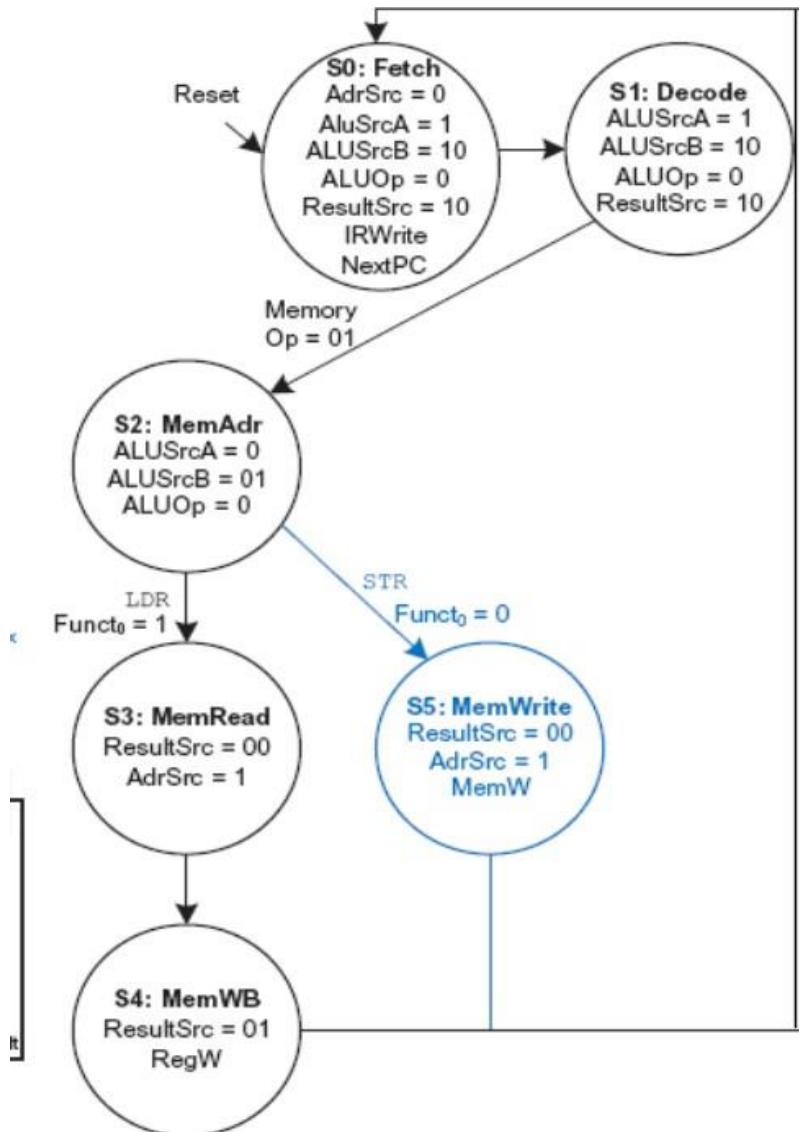


Dallo stato MemAdr, se l'istruzione è di tipo STR (funct₀=0) il dato letto dalla seconda porta del register file deve essere semplicemente scritto in memoria.

Si va nello stato MemWrite.

ResultSrc=00 per selezionare l'indirizzo proveniente dall'alu e AdrSrc=1 per selezionare l'indirizzo calcolato nello stato MemAdr e salvato in registro ALUout

MemW viene attivato per scrivere in memoria e la FSM torna nello stato di fetch.



Dataflow data processing

Per le istruzioni di elaborazione dati (con Op=00) il processore multiciclo deve calcolare il risultato usando l'ALU e scriverlo nel register file.

Il primo operando sorgente viene sempre da un registro (ALUSrcA=0)

ALUOp=1 in modo che il decoder dell'alu possa selezionare il valore appropriato di ALUControl per la specifica istruzione usando il campo cmd.

Il secondo operando sorgente proviene dal register file per istruzioni con modo di indirizzamento a registro (ALUSrcB=00) oppure da ExtImm per istruzioni con modo di indirizzamento immediato (ALUSrcB=01)

La FSM ha bisogno dei due stati ExecuteR e Executel per gestire le due diverse situazioni. In entrambi casi poi l'istruzione avanza allo stato ALUWB di scrittura del risultato dell'ALU, nel quale il risultato del calcolo viene selezionato da ALUOut (ResultSrc=00) e scritto nel register file (RegW=1)

Dataflow branch

Per l'istruzione di salto il processore deve calcolare l'indirizzo di destinazione ($PC+8+offset$) e scriverlo nel PC. Durante la fase di Decode $PC+8$ è già stato calcolato e portato al register file. Quindi nello stato Branch l'unità di controllo setta $ALUSrcA=0$ per selezionare R15 ($PC+8$), $ALUSrcB=01$ per selezionare ExtImm

$ALUOp=0$ per la somma con $ALUControl=00$

Il mux che produce Result seleziona ALUResult (quindi $ResultSrc=10$).

Branch è attivato per scrivere il risultato nel pc

