# Architetture Degli Elaboratori

## Lezione 4
## “Costruire” le istruzioni & ARM Assembly

# Ripasso

## L’architettura di Von Neumann

**Nella sua forma “di base” un computer può essere astratto in 4 componenti**

**CPU**

Una **CPU** o unità centrale di elaborazioni che **esegue** le **istruzioni**

**Memoria primaria**

Una **memoria primaria** che contiene sia le **istruzioni** che i **dati** su cui esse lavorano

Un **Bus** per permettere la **comunicazione** tra le diverse component → collo di botiglia?

**Periferiche**

Un insieme di **periferiche**, che vanno dalla memoria di **massa** (dischi fissi, SSD), ai dispositivi di **input** (tastiere, mouse), ai dispostivi di **output** (monitor, stampanti, etc).

# Dove siamo

**Istruzioni**

Sappiamo:

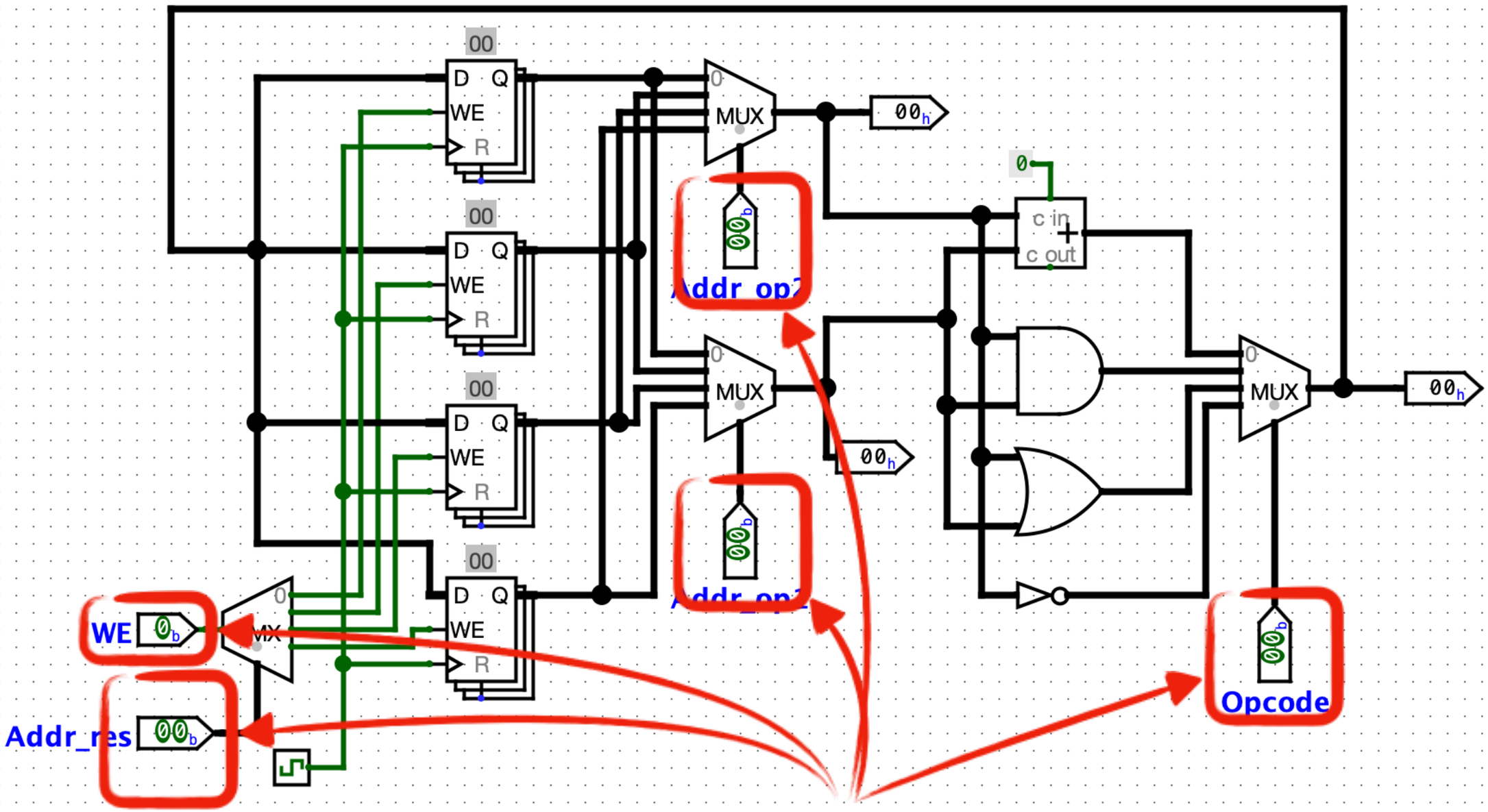
- Rappresentare informazioni in binario
- Fare operazioni in binario
- Instradare dati
- Salvarli permanentemente

**D**: Cosa ci manca?

- Rapresentare le istruzioni

# Register File e ALU

## Cosa abbiamo costruito la volta scorsa




Controllando questi bit possiamo dire che operandi usare,
cosa deve fare la ALU e dove salvare il risultato

# Register File e ALU

## Istruzioni

Teniamo (per ora) Write Enable sempre a 1

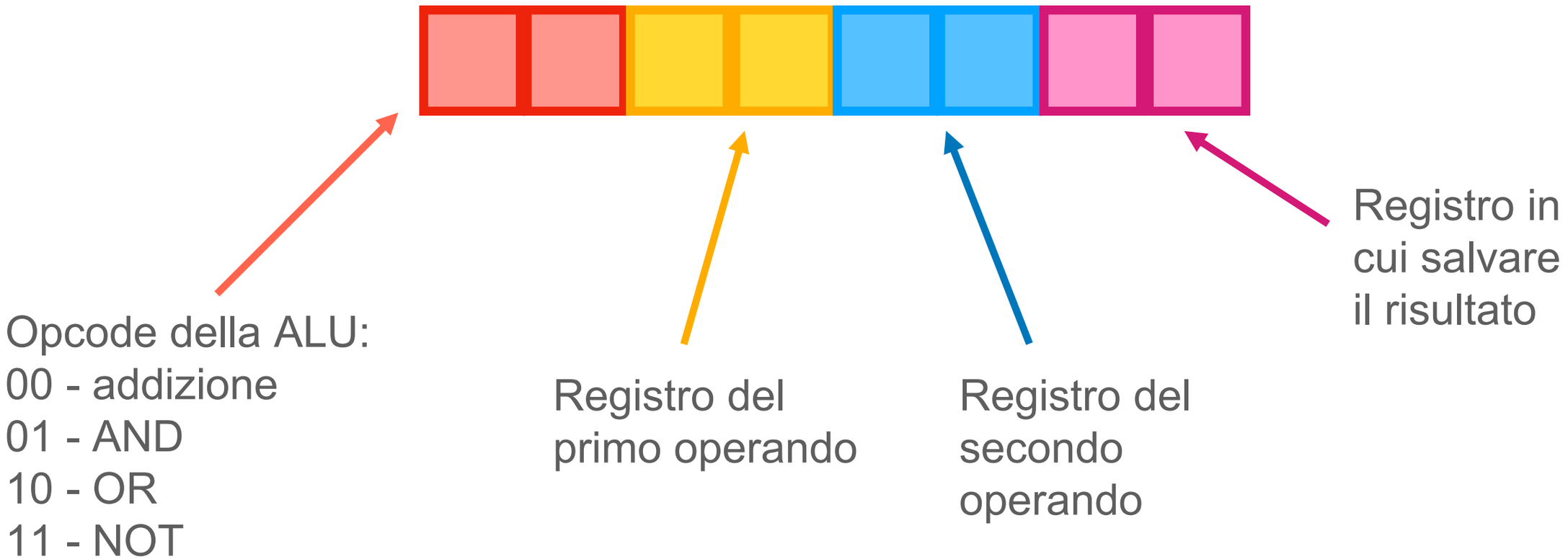
Ci servono 8 bit per controllare:
- Da che registri leggere (2 bit ciascuno)
- Su che registro scrivere (2 bit)
- Che operazione far fare alla ALU (2 bit)

Possiamo codificare il “cosa fare” un istruzione in 8 bit:

# Register File e ALU

## Codifica delle istruzioni

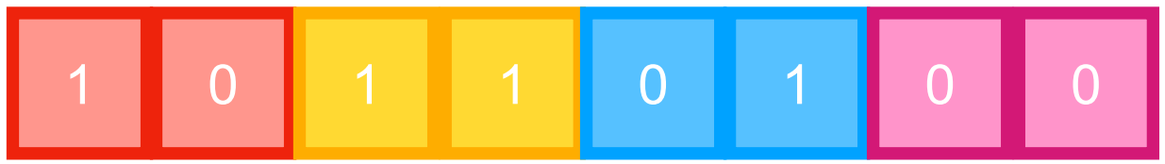
Come mettano i bit è arbitrario, proviamo una possibile codifica



# Register File e ALU

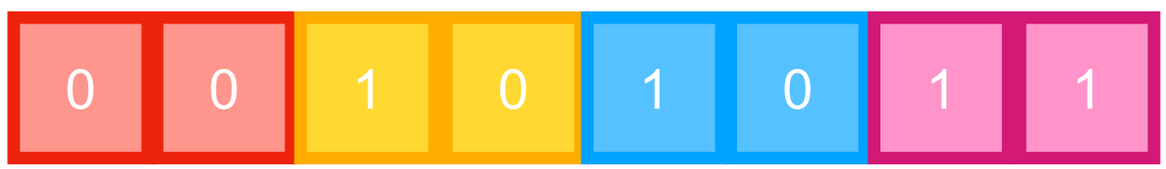
## Codifica delle istruzioni

Cosa sarebbe una possibile istruzione?




B4

Fai OR del contenuto dei registri 3 e 1 salvando il risultato nel registro 0




2B

Fai la SOMMA del contenuto dei registri 2 e 2 e salva il risultato nel registro 3

# Scrivere le istruzioni

## O “ho inventato l’assembly”

- Scrivere 00101011 per “fai la SOMMA del contenuto dei registri 2 e 2 e salva il risultato nel registro 3” è scomodo
- Potremmo usare qualcosa di più leggibile e poi convertire in binario
- Esempio:
  - ADD R3, R2, R2 per “fai la SOMMA del contenuto dei registri 2 e 2 e salva il risultato nel registro 3”
  - AND R0, R3, R1 per “Fai AND del contenuto dei registri 3 e 1 salvando il risultato nel registro 0”

# Dove siamo

## Istruzioni

Sappiamo:

- Rappresentare informazioni in binario
- Fare operazioni in binario
- Instradare dati
- Salvarli permanentemente

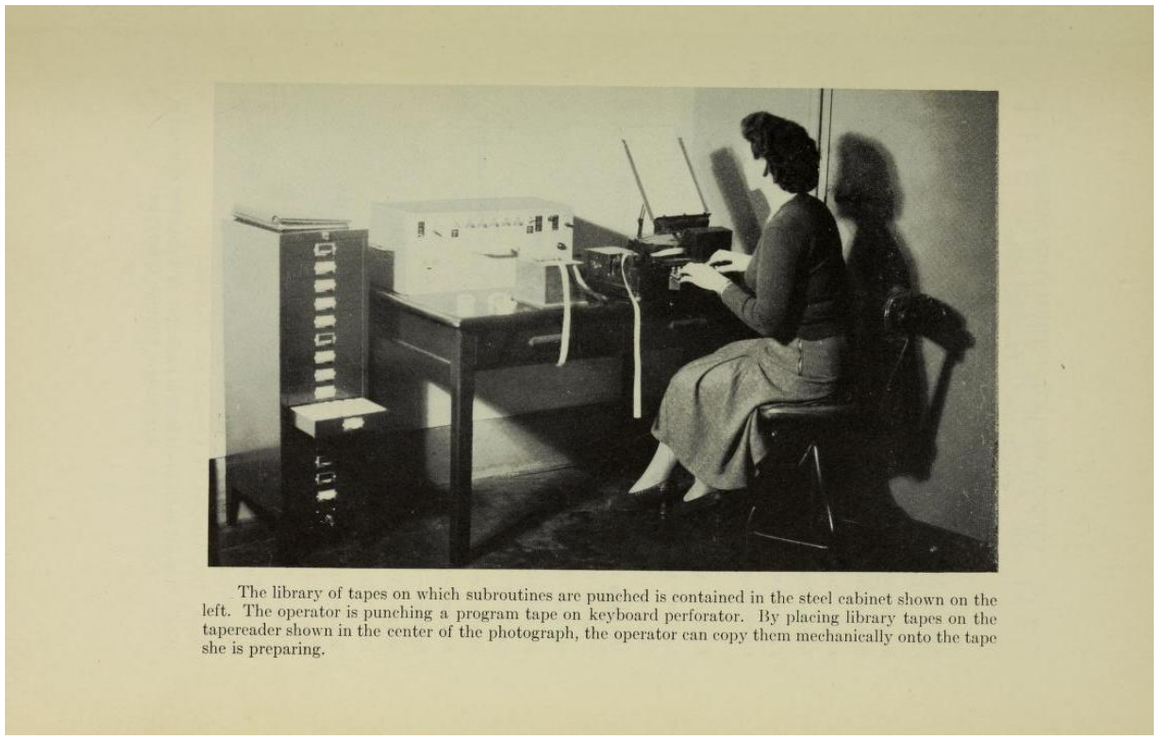
- Rapresentare le istruzioni…
- … che sfruttano le funzionalità descritte sopra

ADD R0, R3, R1

# Scrivere le istruzioni

## O “ho inventato l’assembly”

- Così facendo abbiamo ideato un linguaggio **assembly** per il nostro (ancora non completo) processore
- Se scriviamo un programma che ci fa la conversione automatica da testo a codice binario allora abbiamo costruito un **assembler**



The library of tapes on which subroutines are punched is contained in the steel cabinet shown on the left. The operator is punching a program tape on keyboard perforator. By placing library tapes on the tapereader shown in the center of the photograph, the operator can copy them mechanically onto the tape she is preparing.

Welcome to 1951!

In “The Preparation of Programs for an Electronic Digital Computer” viene introdotto il concetto di assembler per il computer EDSAC

# Scrivere le istruzioni

## O “ho inventato l’assembly”

«Il nastro di carta utilizzato per l’immissione dei dati viene preparato mediante una perforatrice a tastiera. Sul nastro vi sono cinque posizioni, disposte in larghezza, nelle quali possono essere praticati o meno dei fori; una riga di fori può dunque considerarsi come la rappresentazione di un numero binario a cinque cifre.
La perforatrice a tastiera è munita di 32 tasti contrassegnati con lettere, cifre e altri simboli, analogamente a quanto avviene per una comune tastiera di telescrivente.
Ogni pressione di tasto provoca la perforazione di una riga di fori sul nastro, secondo il codice riportato nell’Appendice 1. Nella medesima appendice sono altresì indicati i corrispondenti numeri binari a cinque cifre.»

The Preparation of Programs for an Electronic Digital Computer, §4.2 Input of numbers

# Eseguire più istruzioni

## E dove memorizzarle

- Ora possiamo eseguire una istruzione
- E se ne volessimo eseguire di più in sequenza?
- Potremmo avere un insieme di registri (o una memoria) con indirizzi da 0 a N-1 (il numero di istruzioni che vogliamo eseguire)…
- Ed eseguire prima quella in posizione 0, poi quella in posizione 1, etc.
- Servirebbe un altro registro per dirci quale istruzione eseguire

# Eseguire più istruzioni

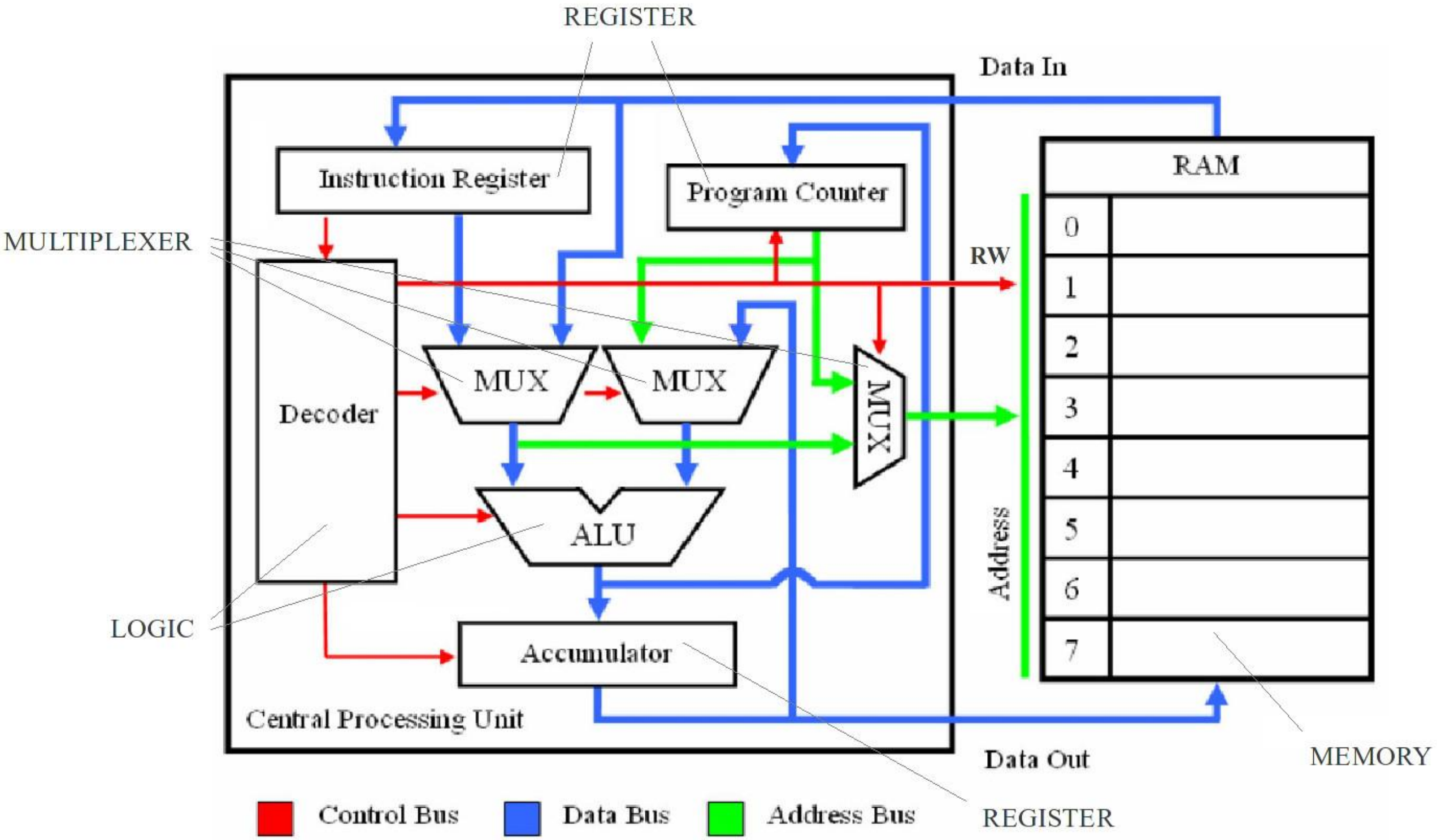
## E dove memorizzarle

- Teniamo traccia in un registro dell'indirizzo dell'istruzione da eseguire
- Dopo aver eseguito una istruzione lo incrementiamo
- Quindi "conta" quante istruzioni del programma abbiamo eseguito
- In questo modo possiamo eseguire una sequenza di istruzioni lunga quanto vogliamo…
- …ma ci manca ancora un po' per un processore generico

# Una semplice CPU

- Design su: http://simplecpudesign.com/simple_cpu_v1/index.html
- Rappresenta una CPU completa con istruzioni di comparazione, salti (permettendo di cambiare il valore del program counter) e accesso alla memoria
- Un solo registro (chiamato “accumulatore”)
- Istruzioni eseguite in 4 cicli di clock:
  - Fetch, decode e execute (spezzata in esecuzione dell’istruzione  e incremento del program counter)

# Una semplice CPU



# Una semplice CPU

Forma delle istruzioni:

- **Load ACC kk** : `0000 XXXX KKKKKKKK`
  Salva nell'accumulatore la costante di 8 bit KKKKKKKK

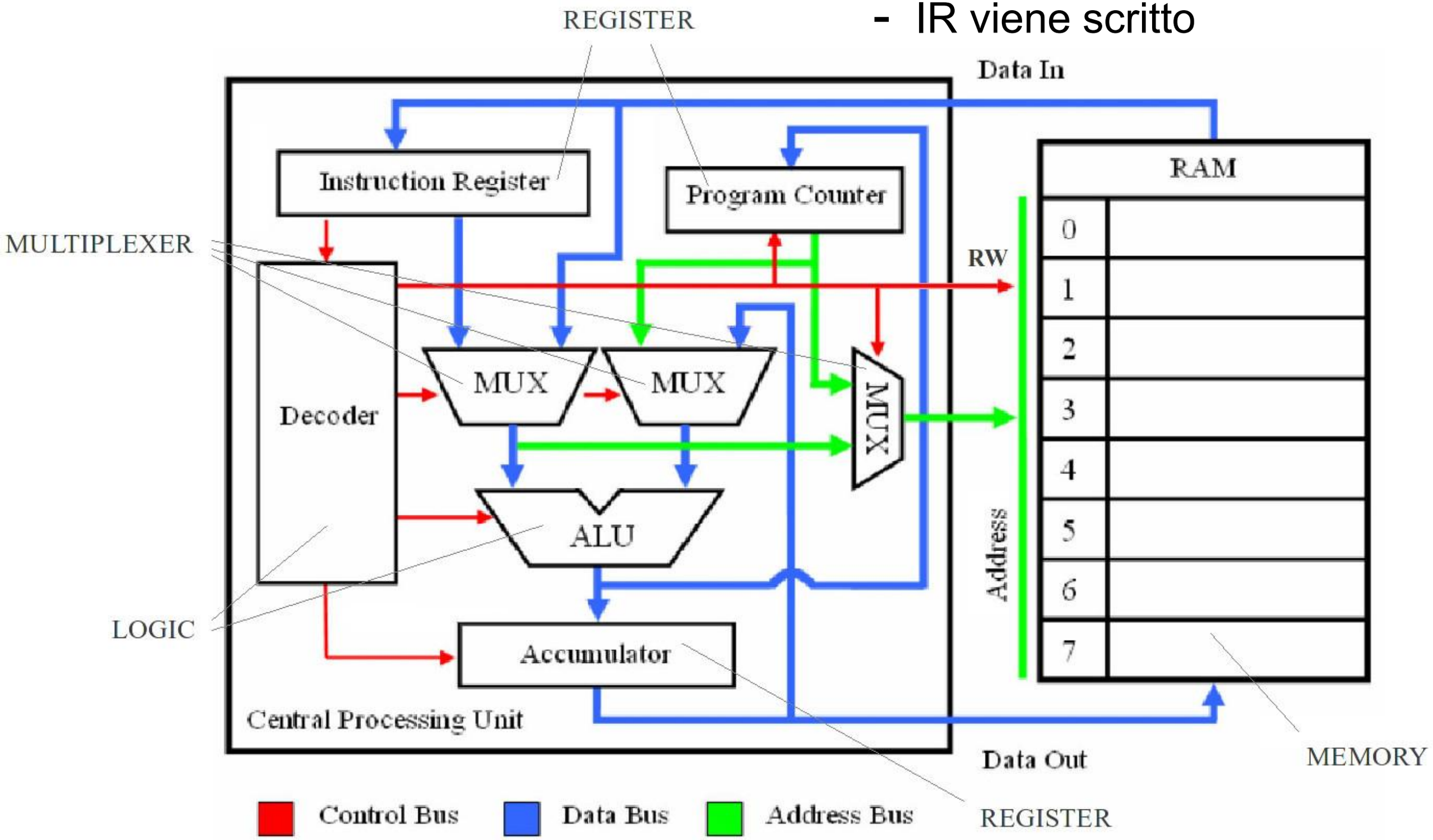
- **Add ACC kk** : `0100 XXXX KKKKKKKK`
  Somma la costante KKKKKKKK al valore contenuto nell'accumulatore

- **Jump Z aa** : `1001 00XX AAAAAAAA`
  Se il valore nell'accumulatore è zero, imposta il program counter all'indirizzo AAAAAAAA

**Architettura del Set di Istruzioni** minimale

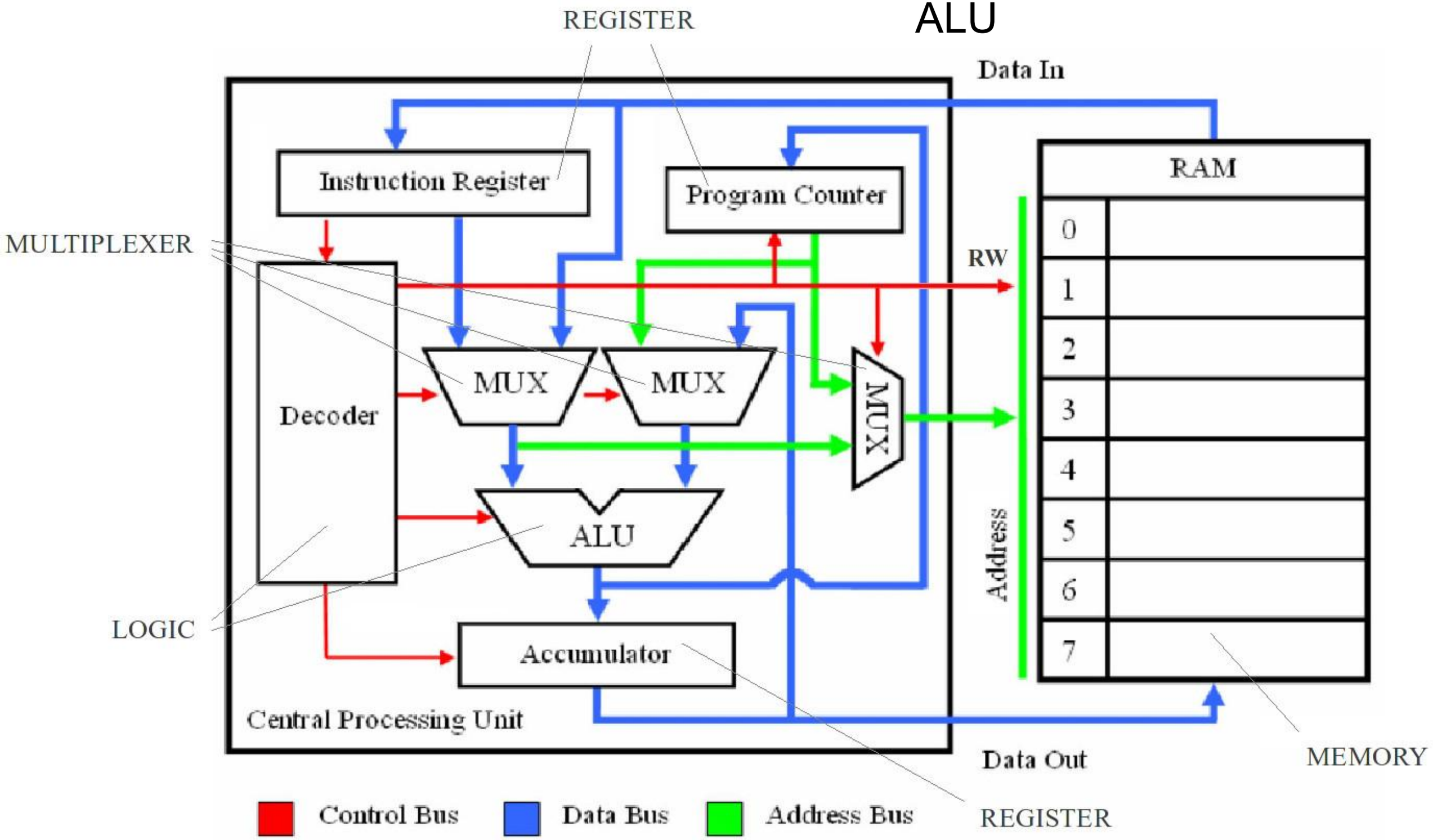
# Una semplice CPU

Fetch:
- PC va in address bus
- RAM mette istruzione su data bus
- IR viene scritto



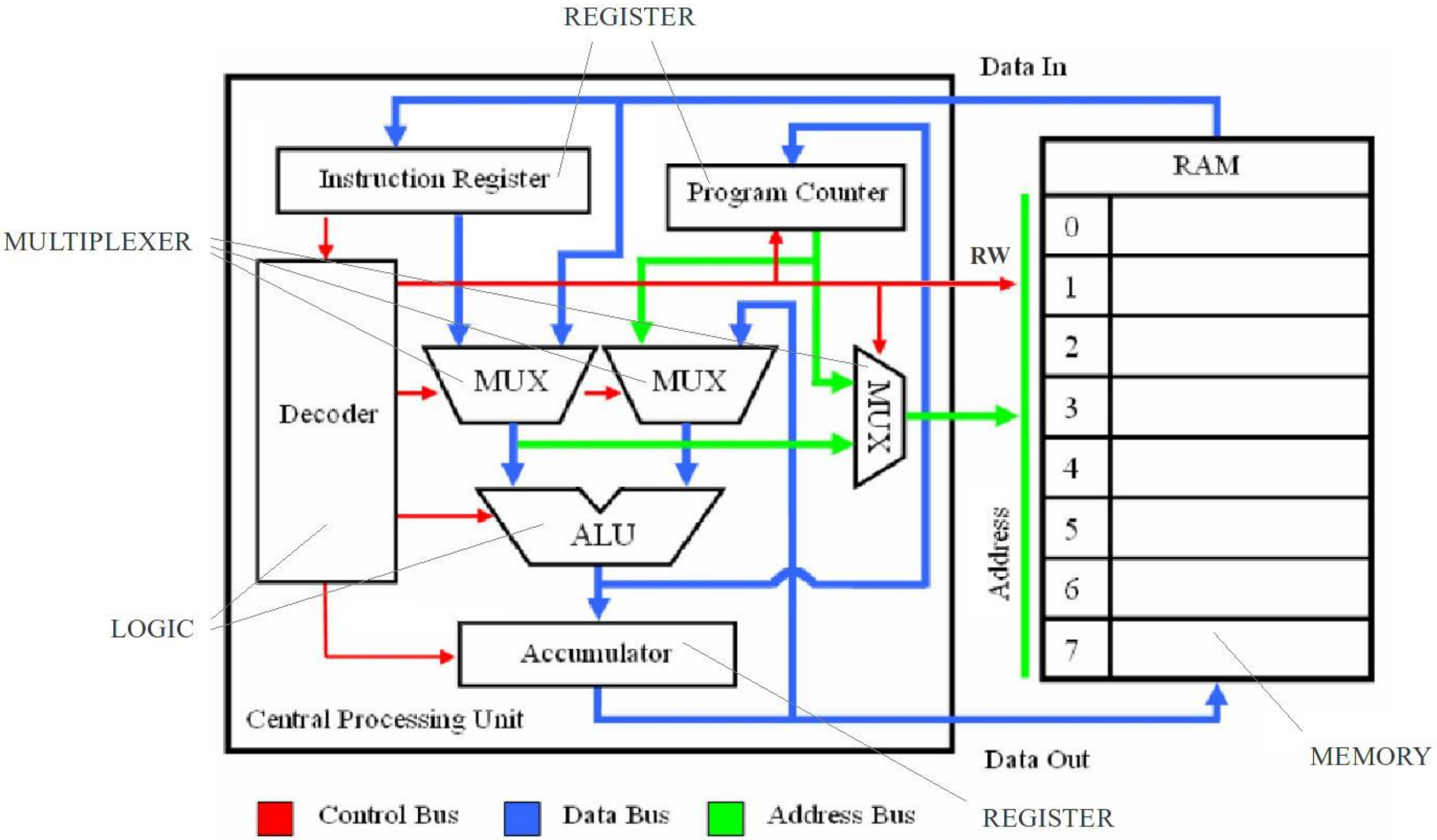
# Una semplice CPU

Decode:
- IR va in decoder
- Decoder attiva segnali in control bus per MUX e ALU



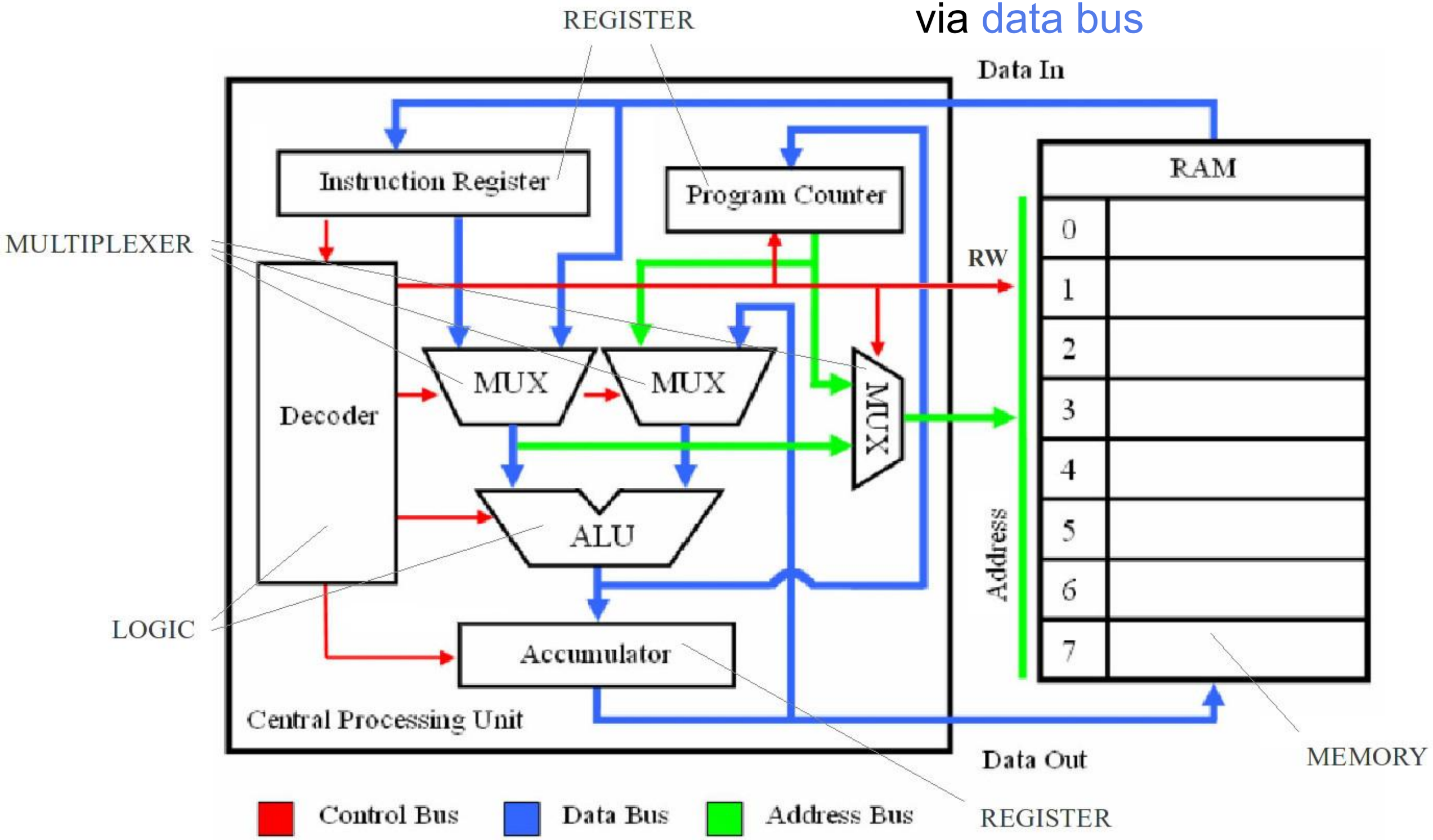
# Una semplice CPU

Execute:
- Aggiornare registro di output (accumulator)



# Una semplice CPU

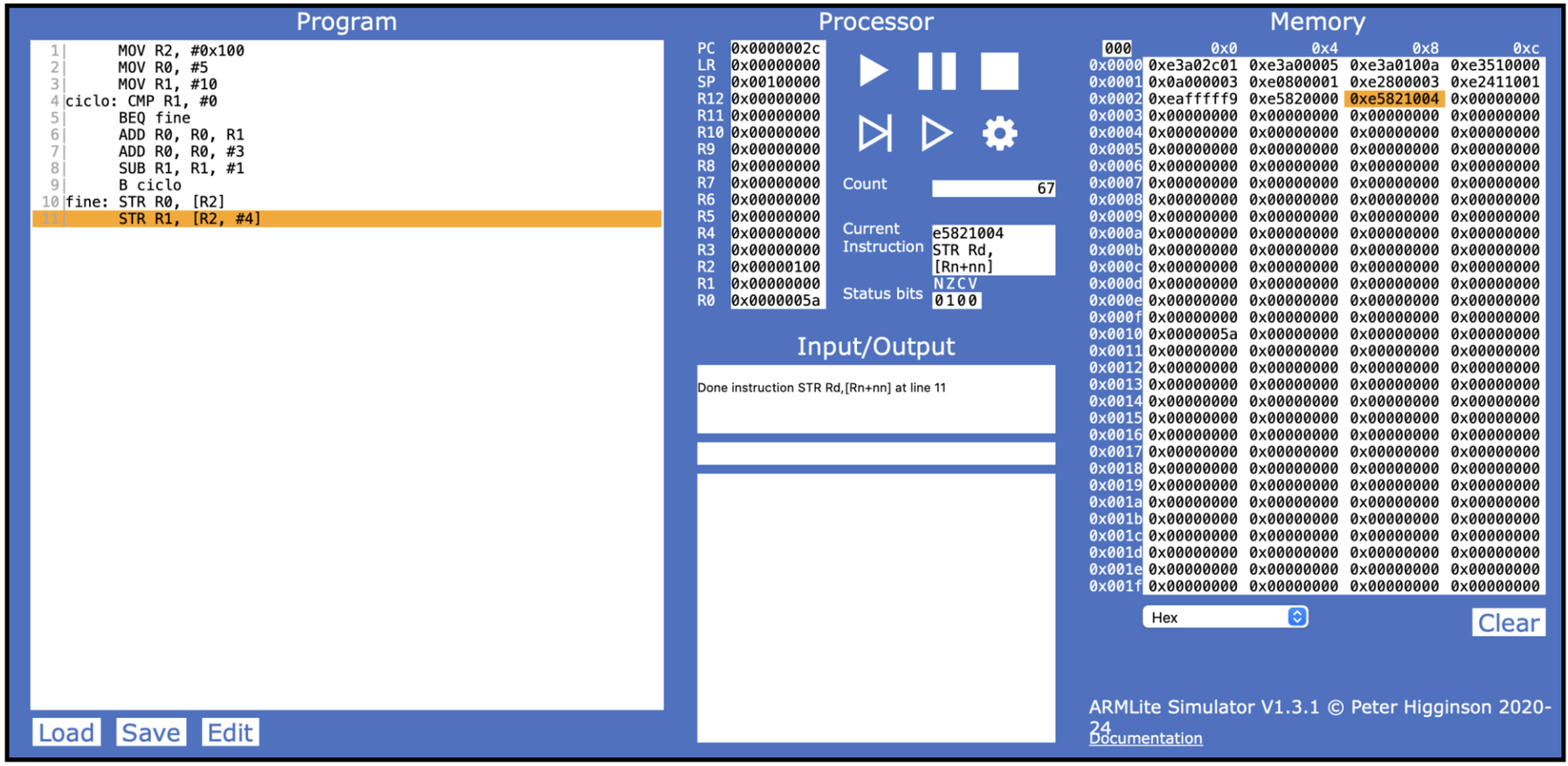
Increment PC:
- Passare PC ad ALU via address bus
- Passare output ALU a PC via data bus



# ARM Assembly

# Software

## ARMLite Simulator




- Il software che utilizzeremo è ARMLite Simulator: https://peterhigginson.co.uk/ARMlite/
- Supporta un sottoinsieme delle istruzioni ARM a 32 bit
- Utilizzabile interamente online
- Permette di eseguire le istruzioni passo passo e visualizzare il contenuto di registri e memoria

# La storia di ARM

## Da Acorn Risc Machine a ARM

- L'architettura ARM nasce come prodotto della Acorn Computers, società britannica che aveva prodotto il **BBC Micro** nel 1981, un **home computer**

- Dopo il BBC Micro e l'Acorn Electron, la Acorn decise di sviluppare una **architettura propria** di processori con il progetto ***Acorn RISC Machine*** **(ARM)**. Nel 1985 l'ARM1 fu il primo processore prodotto

- Nel 1990 la progettazione dei processori fu separata in una diversa **compagnia** (ARM) fondata da Acorn, Apple e VLSI Technology

- ARM attualmente non produce direttamente processori ma li progetta e vende i diritti di **proprietà intellettuale**

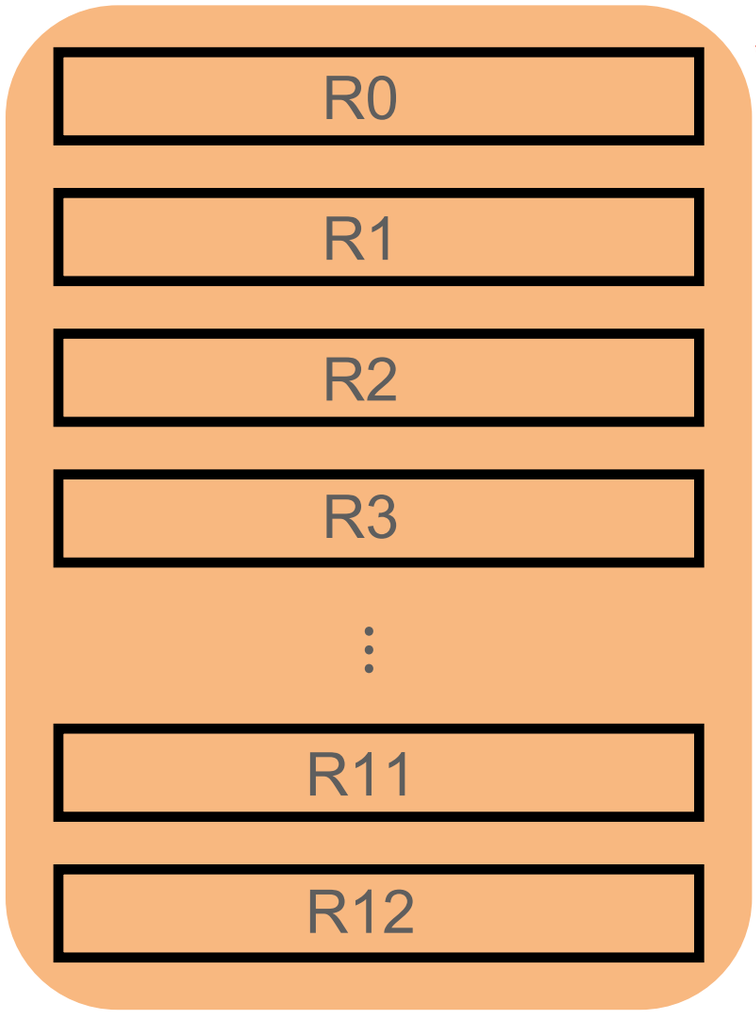
- Nome: Acorn Risc Machine → **Advanced Risc Machine** → ARM

# Arm 32 bit

## Arm-v7a

- 13 **registri** per l'**uso generale**, ognuno di 32 bit
- 1 registro per lo **stack pointer**
- 1 **link register** (per le chiamate a subroutine)
- 1 **program counter**
- 1 **program status register** (PSR)
- Ogni istruzione richiede 32 bit
- Permette una modalità "ridotta" (Thumb mode) in cui ogni istruzione è a 16 bit (che non vedremo)

# Arm: registri




Viene salvato l'indirizzo dello stack (quello delle chiamate a funzione)

R13 (o SP)

Link register viene usato per salvare un indirizzo per alcune operazioni

LR

Contiene l'indirizzo dell'istruzione da eseguire

PC

Contiene diversi bit che vengono impostati come risultato di comparazioni, operazioni aritmetiche, etc

PSR

Nessuno di questi registri ha un utilizzo specifico, possiamo usarli come vogliamo*

* esistono delle convenzioni quando si lavora con altro codice, ma sono solo convenzioni

# Arm 64 bit

## Arm-v8a

- Il passaggio ai 64 bit è stato l'occasione per rendere più consistenti le istruzioni (ma è ancora riconoscibile come ARM)
- 31 registri per uso generale, ognuno da 64 bit (accessibili anche come registri a 32 bit)
- Program Counter non più direttamente accessibile
- Le **istruzioni** rimangono codificate in **32 bit**

# CISC vs RISC

# CISC

## Complex Instruction Set Computers

- Nell'approccio CISC si vuole completare un'operazione nel **minor numero di operazioni** in assembly

- Questo significa che, per esempio, è possibile nella stessa operazione accedere alla memoria, sommare un valore, scrivere il valore in memoria

- ADD [mem1], VAL ; somma VAL al valore in locazione mem1

vs

- MOV R0, #mem1 ; scrivi locazione in registro
- LDR R1, [R0] ; carica da memoria in R1
- ADD R1, R1, #VAL ; somma R1 e VAL
- STR R1, [R0] ; salva in memoria

# CISC

## Complex Instruction Set Computers

- Nell'approccio CISC si vuole completare un'operazione nel **minor numero di operazioni** in assembly
- Questo significa che, per esempio, è possibile nella stessa operazione accedere alla memoria, sommare un valore, scrivere il valore in memoria
- Generalmente le istruzioni richiedono **più di un ciclo di clock** per essere completate
- Generalmente le istruzioni sono di **lunghezza variabile**, sono **molte** e più **difficili** da decodificare
- Esempio: x86

# RISC

## Reduced Instruction Set Computers

- Nell'approccio RISC si vuole mantenere **basso** il **numero** di **istruzioni**
- Ogni **istruzione** compie **una cosa sola**. Si hanno istruzioni separate per caricare dalla memoria in un registro, sommare un valore e salvare il risultato in memoria
  - **Svantaggio**: Devo scrivere tante istruzioni assembly: stesso programma per architetture RISC occupa più spazio di un CISC
- Si tende ad avere istruzioni che completano in **pochi cicli di clock**
- Le istruzioni sono di **lunghezza fissata**, rendendo la **decodifica** più **semplice**
- In realtà le architetture RISC sono andate complicandosi e quelle CISC hanno preso molte idee dai RISC → **differenza sfumata**.
- Esempio: ARM

# Istruzioni ARM

# Assembly

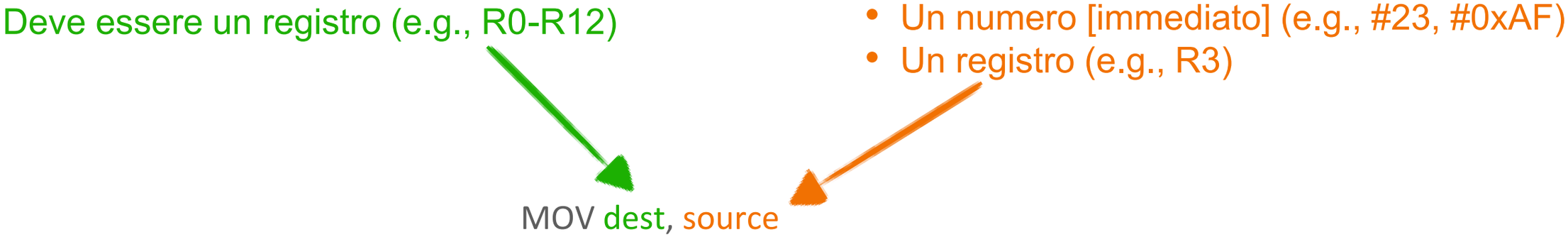
## Sarà uguale a quello visto prima?

- Le istruzioni, come abbiamo visto, sono  codificate come sequenze di bit
- Questa rappresentazione testuale che ha una conversione diretta (o, comunque, semplice) alle istruzioni in codice macchina è detta, come avviamo visto, **assembly**
- Notate però come l'assembly dipenda dall'architettura del processore…
- …se abbiamo più operazioni e più registri possibili ci aspettiamo di vedere istruzioni diverse da quelle viste prima!

# MOV

## Spostare valori tra registri

L’istruzione MOV permette di copiare un valore da un registro a un altro
o caricare un valore numerico (chiamato immediato) nel registro




**Alcuni esempi di istruzione MOV:**

MOV R0, #45

Mette il valore 45 nel registro R0

MOV R0, R2

Copia il valore contenuto nel registro R2 nel registro R0

**Esercizio:** scambiare il valore del registro R1 e R2 usando R0 come registro di appoggio

# Domande varie su memoria

**D1**: una cifra esadecimale quanti bit rappresenta?

**D2**: quanti bit contiene una cella di memoria nel simulatore?

**D3**: quanti bit occupa un'istruzione in memoria?

**D4**: cosa significa 0x4?

```
MOV R0, #3
MOV R1, #7
MOV R10, #9
```

**D5**: dove mi aspetto la terza istruzione?

**D6**: quali bit codificano MOV, Rx, #y?

# ADD

## Sommare Valori

L’istruzione ADD somma i valori contenuti in due registri o il valore contenuto in un registro e un numero.
Il risultato viene poi salvato in un registro


**Alcuni esempi di istruzione ADD:**

ADD R0, R0, #1

Incrementa il valore contenuto in R0 di 1

ADD R0, R1, R2

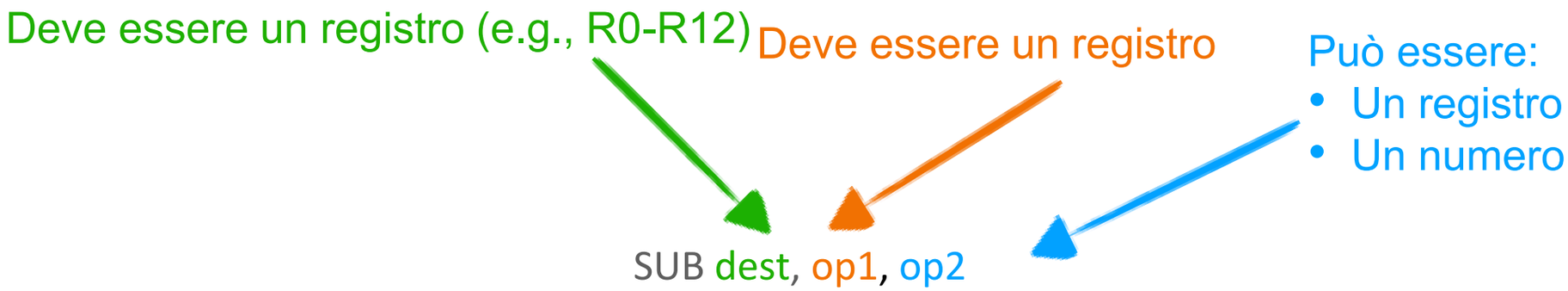
Mette in R0 il risultato della somma dei valori contenuti in R1 e R2

**Esercizio:** Mettere in R1 un valore a scelta. Usando istruzioni ADD mettere in R0 il triplo del valore contenuto in R1

# SUB e altre istruzioni

## ADC, SBC, RSB, RSC, AND, ORR, EOR, BIC, ORN

L'istruzione SUB sottrae i valori contenuti in due registri o il valore contenuto in un registro e un numero.
Il risultato viene poi salvato in un registro



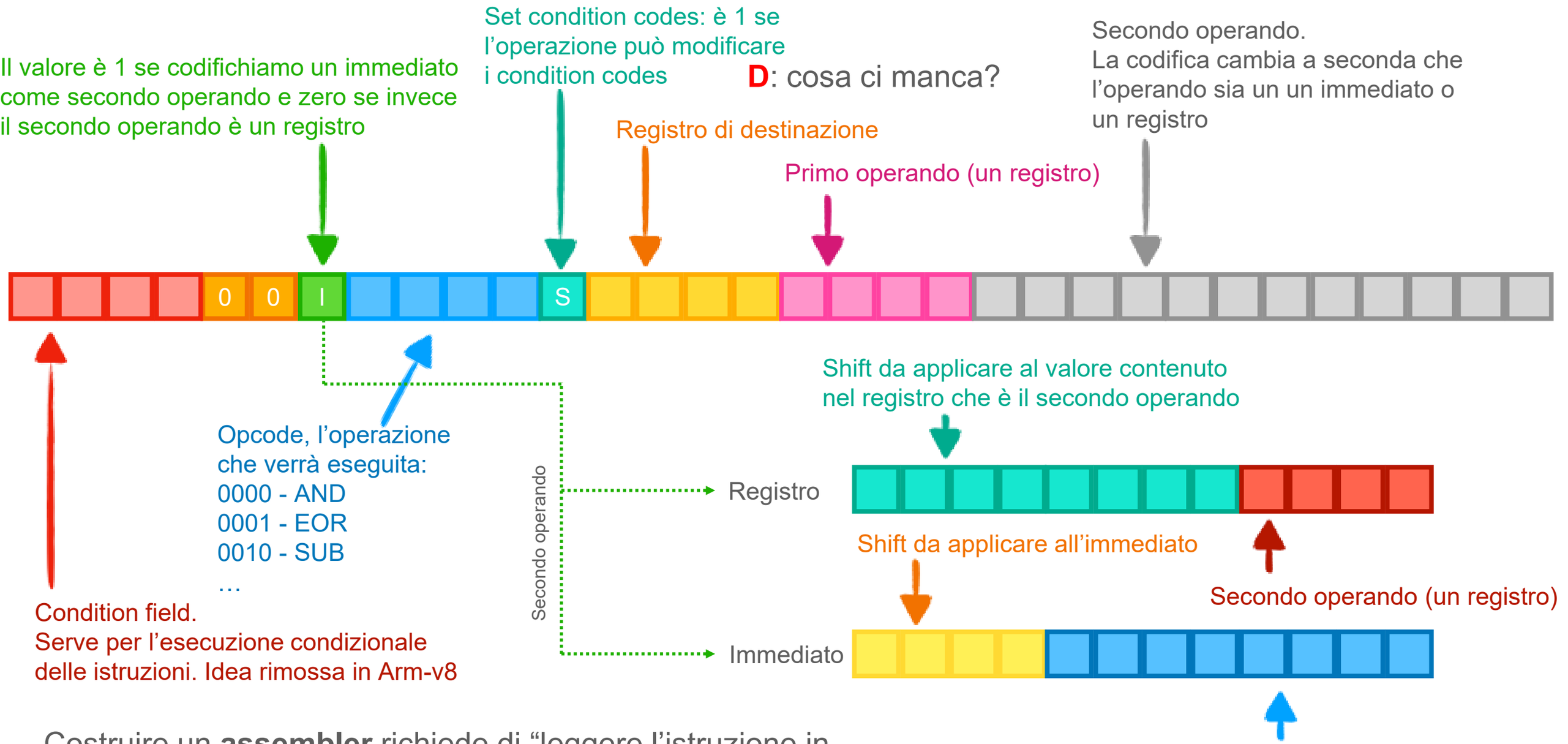

**Altre istruzioni con formato simile:**

| | | | |
|---|---|---|---|
| AND dest, op1, op2 | AND bit a bit | LSL dest, op1, #op2 | Left bit shift by op2 |
| ORR dest, op1, op2 | OR bit a bit | LSR dest, op1, #op2 | Right bit shift by op2 |
| EOR dest, op1, op2 | Exclusive OR | | |

# Codifica delle istruzioni

## 32 bit per istruzione

Abbiamo detto che ogni istruzione è codificata in 32 bit, vediamo un esempio per le istruzioni cosiddette di “Data Processing” (e.g., ADD, SUB, etc.)




Costruire un **assembler** richiede di “leggere l’istruzione in assembly” e **tradurre in binario secondo queste regole**

# Effettuare scelte

## Istruzioni di salto (jump o branch)

- Fino ad ora abbiamo visto come fare operazioni aritmetiche o logiche
- Non possiamo ancora alterare il flusso di esecuzione delle istruzioni (if/for/while)

**D**: che registro dovremo modificare?

- Per fare questo abbiamo le istruzioni di salto (o branch) che **cambiano** il valore del **program counter**. Distinguiamo due categorie:
  - Salto **non condizionato** avviene sempre
  - Salto **condizionato** avviene solo se alcune condizioni sono soddisfatte (si utilizza il program status register)

# B, BL ed etichette

## Branch e labels

L’istruzione di branch ci permette di saltare ad un’altra istruzione nel codice. Per indicare a quale istruzione dobbiamo saltare usiamo, in assembly, della label.

Deve essere una etichetta
(un nome che precede una istruzione)

B label

**Branch with link**
Funziona come branch ma salva il valore del PC nel link register

BL label

etichetta

```
inizio: ADD R0, R0, #1
B inizio
```

Continuerà a sommare 1 al valore contenuto nel registro R0

Senza i salti condizionati e usando solo istruzioni di branch otterremmo solo cicli infiniti

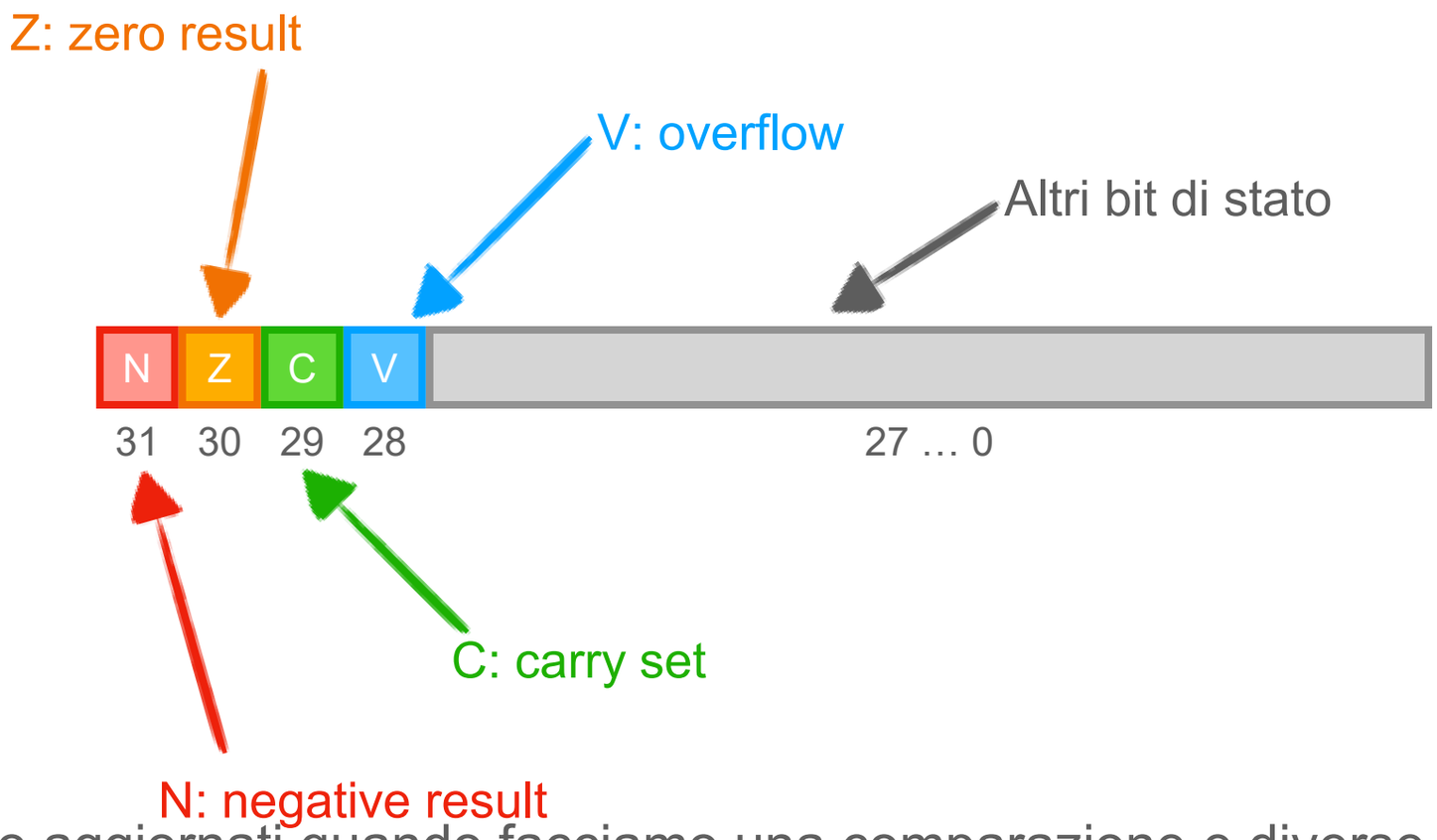
**Codifica:**

Link bit (distingue Branch da Branch with link)

Offset: di quanto  modificare il program counter (questo valore viene calcolato dall’assembler)

# Program Status Register (PSR)

## Ci parla dell'ultimo risultato ALU



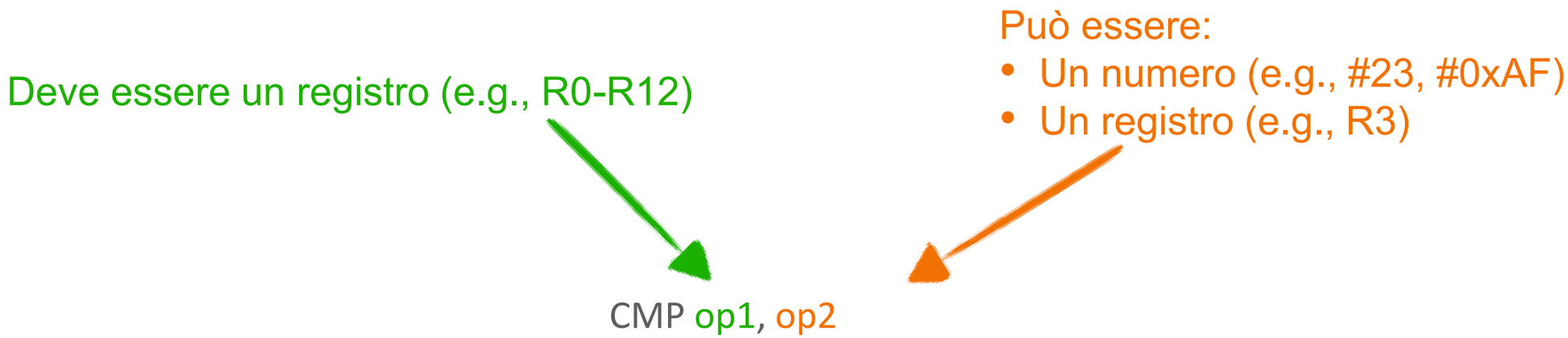

Questi bit sono aggiornati quando facciamo una comparazione e diverse configurazioni di questi bit ci forniscono informazioni sul risultato della comparazione

- Calcolo A - B in ALU
- ALU dà Z=0 e N=0
- Allora (A > B) è True

# CMP

## Comparare valori

L’istruzione CMP ci permette di comparare il valore contenuto in un registro con il valore contenuto in un altro registro o con un immediato. **Il risultato della comparazione altererà il PSR, il valore non si salva**




**Alcuni esempi di istruzione CMP:**

CMP R0, #45

Compara il valore contenuto nel registro R0 con il valore 45

CMP R0, R2

Compara il valore contenuto nel registro R0 con quello contenuto nel registro R2

**Una operazione CMP cambia solo i valori contenuti nel PSR**

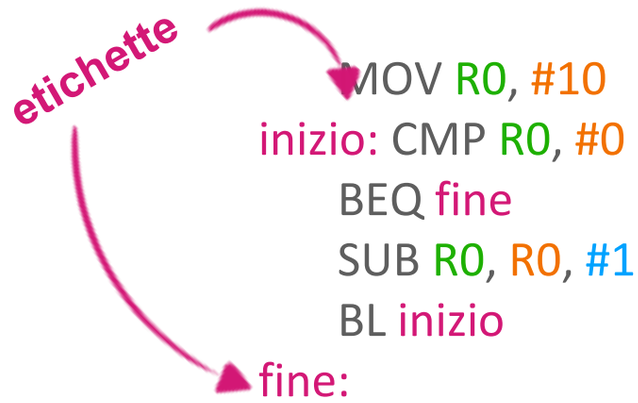
# Branch condizionale

## BEQ

L’istruzione di branch condizionale ci permette di saltare ad un’altra istruzione nel codice.
Con BEQ questo accade solo se l’ultima comparazione (con CMP) era tra due valori identici

Deve essere una etichetta
(un nome che precede una istruzione)

BEQ label




Sottrae uno al valore contenuto in R0
Continua a eseguire l’operazione a meno che
il valore in R0 non sia 0

**Codifica:**

Cambiano solo i bit nel condition field
(rispetto al normale branch)

# Altri tipi di branch condizionale

## BNE, BHS, BLO, …

A seconda dei bit nel condition field si catturano diverse condizioni, corrispondenti a diverse istruzioni in Assembly

| | | |
|---|---|---|
| | BNE label | I due valori comparati erano diversi |
| Senza segno | BHI label | Il primo valore era > del secondo (considerati senza segno) |
| | BLO label | Il primo valore era < del secondo (considerati senza segno) |
| | BHS label | Il primo valore era ≥ del secondo (considerati senza segno) |
| | BLS label | Il primo valore era ≤ del secondo (considerati senza segno) |
| Con segno | BGT label | Il primo valore era > del secondo (considerati con segno) |
| | BLT label | Il primo valore era < del secondo (considerati con segno) |
| | BGE label | Il primo valore era ≥ del secondo (considerati con segno) |
| | BLE label | Il primo valore era ≤ del secondo (considerati con segno) |

# Architetture Load-Store

## Accedere alla memoria di lavoro

- Se notate nessuna delle operazioni precedenti utilizza direttamente sulla memoria di lavoro
- Sono solo operazioni tra registri
- Se vogliamo fare **operazioni su** valori salvati nella **memoria** di lavoro dobbiamo prima **copiare** il valore **in** un **registro**
- **ARM** utilizza operazioni esplicite per interagire con la memoria:
  - **LOAD** per caricare dalla memoria in un registro
  - **STORE** per salvare il valore di un registro in memoria

# Architetture Load-Store

## Indirizzamento

- Per accedere alla memoria ci serve un indirizzo
- Se codificassimo l'indirizzo direttamente nell'istruzione potremmo accedere solo a una frazione della memoria (e.g., se avessimo 24 bit "liberi" per l'indirizzo potremmo accedere solo a $2^{24}$ diverse locazioni di memoria, solitamente byte)
- Di solito si **salva** l'**indirizzo** di memoria in un **registro** (quindi 32 bit possibili) e si accedere all'indirizzo indicato da quel registro:
  - e.g., "Leggi in R0 il valore contenuto all'indirizzo di memoria che è scritto in R1"

# LOAD

## Caricare valori dalla memoria

L’istruzione LDR ci permette di caricare in un registro un valore in memoria
Dato che le operazioni come ADD, SUB, … lavorano solo su registri questo è essenziale

Deve essere un registro (e.g., R0-R12)

Un registro che contiene l’indirizzo da cui prendere il valore

LDR dest, [addr]

**Alcuni esempi di istruzione LDR:**

LDR R0, [R1]

Carica in R0 il valore contenuto all’indirizzo di memoria che è il valore di R1

MOV R0,#0x100
LDR R1, [R0]

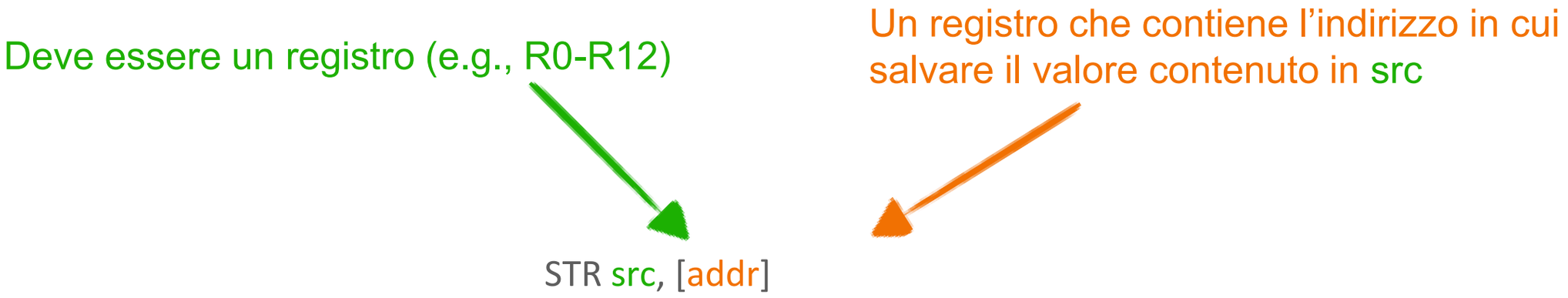
Carica in R1 il valore contenuto nell’indirizzo di memoria 0x100 (indicato in esadecimale)

**Esercizio:** Caricare in R1 il valore contenuto all’indirizzo di memoria 0x200

# STORE

## Salvare i valori in memoria

L’istruzione STR ci permette di salvare in memoria il valore contenuto in un registro.




**Alcuni esempi di istruzione STR:**

STR R0, [R1]

Salva il valore di R0 nell’indirizzo di memoria che è il valore di R1

MOV R0,#0x100
STR R1, [R0]

Salva all’indirizzo 0x100 il valore contenuto nel registro R1

**Esercizio:** Salvare all’indirizzo di memoria 0x1fc il risultato di 12+4

# Load e Store con offset

## Accedere ad array

- È **comune** dover accedere a **valori contigui** in memoria
- Per esempio i valori in un array sono salvati in modo contiguo in memoria
- La struttura degli indirizzi è quindi un “**indirizzo di base**” (primo elemento dell’array) e un **offset** (di quando incrementare l’indirizzo rispetto alla base)
- Questo è direttamente **supportato** dalle istruzioni LDR e STR
- Supportano anche il pre- e post-incremento dell’indirizzo di base del valore indicato dall’offset

# LOAD

## Questa volta con offset

L'offset può essere:
- Un registro (in quel caso viene considerato come valore il contenuto del registro)
- Un immediato (che rappresenta direttamente il valore dell'offset)

LDR dest, [addr, offset]

Accede all'indirizzo addr + offset

LDR dest, [addr, offset]!

**Pre-incremento**: somma offset a addr, salvando il valore in addr, e accede a addr (dopo che il suo valore è stato incrementato di addr)

LDR dest, [addr], offset

**Post-incremento**: accede a addr (prima che il suo valore sia incrementato). Somma addr e offset e salva il suo valore in addr.

# STORE

## Questa volta con offset

L'offset può essere:
- Un registro (in quel caso viene considerato come valore il contenuto del registro)
- Un immediato (che rappresenta direttamente il valore dell'offset)

STR dest, [addr, offset]

Accede all'indirizzo addr + offset

STR dest, [addr, offset]!

**Pre-incremento**: somma offset a addr, salvando il valore in addr, e accede a addr (dopo che il suo valore è stato incrementato di addr)

STR dest, [addr], offset

**Post-incremento**: accede a addr (prima che il suo valore sia incrementato). Somma addr e offset e salva il suo valore in addr.

# Accesso allineato

## E possibili errori con LDR e STR

- Un possibile problema con le operazioni di LOAD e STORE è che gli accessi in memoria devono essere **allineati**
- Cosa significa? La memoria viene letta a **gruppi** di byte **contigui** (nel nostro caso **4 byte** o 32 bit) ma:
  - Gli **accessi** sono **più efficienti** se si legge da multipli di 4 byte (indirizzi allineati)
  - Accessi **non allineati** non sono concessi (in ARM). Altre architetture li concedono, ma possono essere più lenti, richiedendo **due accessi** allineati in memoria
- Ergo, possiamo  accedere solo ad indirizzi che sono multipli di 4

# Esercizio

## Fare il lavoro del compilatore

Facciamo il lavoro di un compilatore: proviamo a convertire questo ciclo in del codice assembly in modo “plausibile”.
Corretto se alla fine x=90

**Due varianti** dell’esercizio:

- x e y sono valori nei registri (e.g., R0 e R1)
- x e y sono da salvare alle locazioni di memoria 0x100 e 0x104

```
int x = 5;
int y = 10;
while (y > 0) {
    x = x + y + 3;
    y = y - 1;
}
```

Notate che dobbiamo spezzare questa operazione di somma in due operazioni distinte!