Dicii

Università di Roma Tor Vergata

# z64: Unità di Processamento

*Alessandro Pellegrini*
*a.pellegrini@ing.uniroma2.it*

# Passi essenziali per la progettazione di una CPU

- Definire le componenti hardware di calcolo interne alla CPU
- Identificare le modalità di interconnessione tra le componenti
- Identificare le componenti esterne alla CPU con cui è necessario trasferire dati
- Scegliere quali istruzioni supportare e codificarle
- Definire la modalità di esecuzione delle istruzioni
- Sintetizzare l'unità di controllo

- Molti di questi passi contribuiscono alla definizione dell'Instruction Set Architecture (ISA) e dell'Application Binary Interface (ABI)
- L'ISA dello z64 è ispirata all'ISA Intel x86
  - problema principale: *retrocompatibilità*

# Instruction Set Architecture (ISA)

- L'ISA è un "contratto" tra progettisti hardware e sviluppatori software
  - insieme delle istruzioni, tipi di dato supportati, formato delle istruzioni

- *Reduced Instruction Set Computers* (RISC):
  - insieme piccolo di istruzioni (programmi più lunghi)
  - più efficiente (prestazioni e consumo energetico)
  - di più semplice progettazione
- *Complex Instruction Set Computers* (CISC):
  - una grande quantità di istruzioni (programmi più compatti)
  - più lento o più energivoro
  - di più complessa progettazione

- Per approfondire: *David Chisnall.* How to Design an ISA. Communications of the ACM 67(5) 2024

# Famiglie principali di architetture

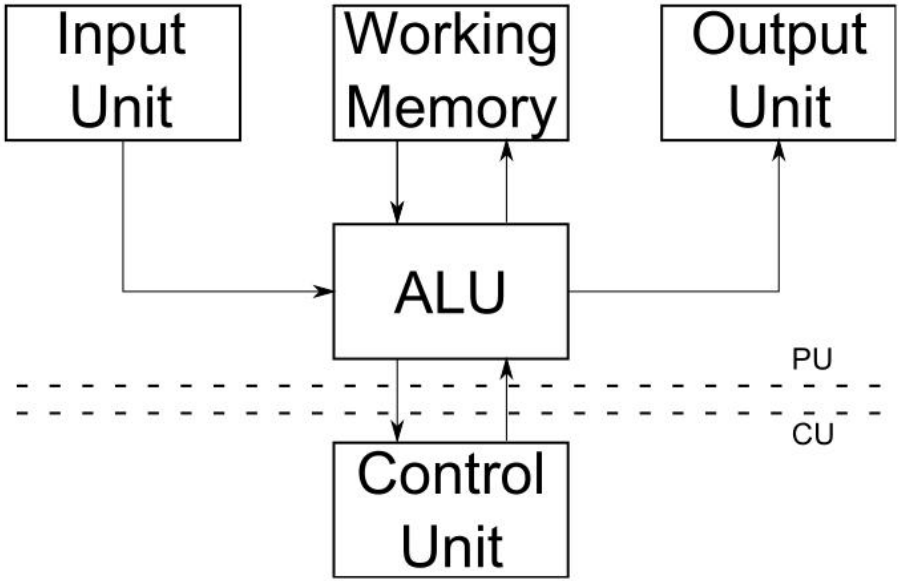
- Nella storia, sono state utilizzate tre famiglie principali di architetture
- A *registri*
  - Il processore contiene una piccola parte di memoria chiamato *banco dei registri*
  - Le operazioni possono operare unicamente su dati contenuti nei registri
- A *stack*
  - I dati vengono "impilati" in memoria
  - Le operazioni possono processare soltanto i dati che si trovano sulla cima della pila
- Ad *accumulatore*
  - Una variante dell'architettura a registri
  - Le operazioni utilizzano sempre uno specifico registro chiamato "accumulatore"
- L'architettura x86 che costruiremo è un ibrido di tutte e tre

# Componenti interne

## e loro interconnessione

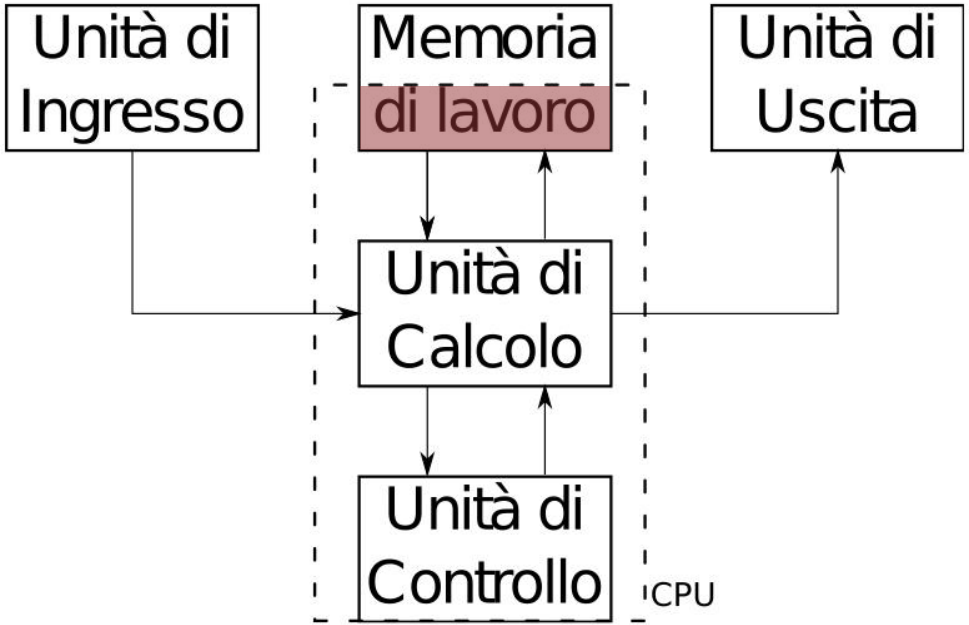
# Architettura di von Neumann rivisitata

- Abbiamo suddiviso logicamente l'archiettura di von Neumann in due blocchi logici:
  - Unità di calcolo: tutto ciò che esegue il lavoro
  - Unità di controllo: la componente in grado di *interpretare* le istruzioni



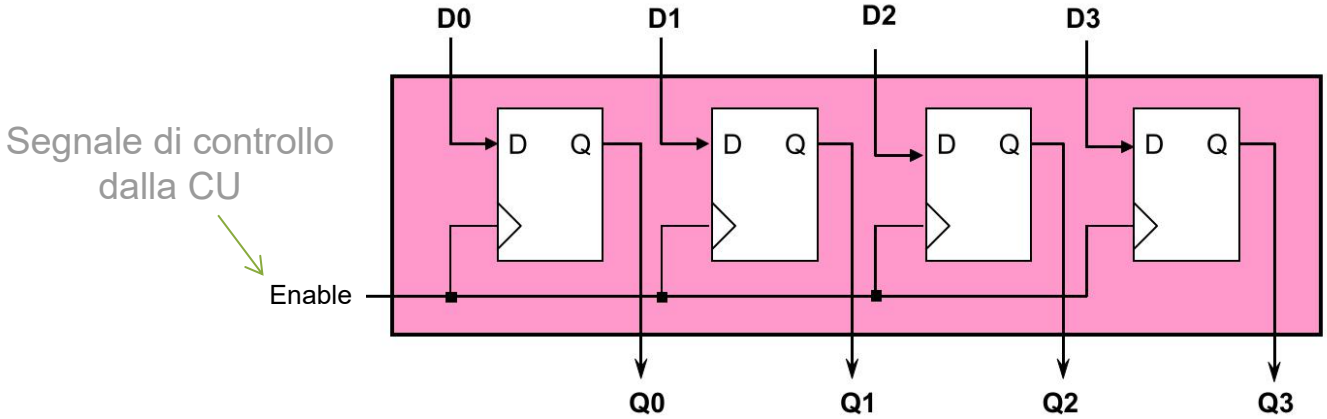
# Architettura di von Neumann rivisitata

- In realtà, all'interno della nostra CPU troviamo:
  - L'unità di controllo
  - L'unità di calcolo
  - *Parte* della memoria di lavoro (architettura a registri)



# I registri

- I registri sono delle unità di memoria interne al processore
  - compongono la parte di memoria di lavoro interna alla CPU
- Permettono di memorizzare *parole binarie*
- La quantità di memoria disponibile è *estremamente limitata*
- Sono realizzati a partire da flip/flop



# I registri

- I registri di una CPU possono essere suddivisi in più classi

- *Registri fondamentali*: sono questi registri senza i quali non è possibile realizzare un'architettura di von Neumann
- *Registri visibili al programmatore*: sono registri che il programmatore può utilizzare esplicitamente nel suo programma
- *Registri invisibili al programmatore*: sono registri che il programmatore può modificare solo indirettamente e non programmaticamente
  - Non è detto che siano esplicitati all'interno dell'ISA

# Registri visibili al programmatore

- Ci sono 16 registri *general purpose* a 64 bit che il programmatore può utilizzare esplicitamente come *operandi* delle istruzioni

| | |
|---|---|
| RAX | R8 |
| RCX | R9 |
| RDX | R10 |
| RBX | R11 |
| RSP | R12 |
| RBP | R13 |
| RSI | R14 |
| RDI | R15 |

Aggiunti da AMD nell'estensione a 64 bit dei processori x86

- Alcuni di questi registri hanno un significato particolare e sono utilizzabili *implicitamente* utilizzando specifiche istruzioni assembly

# Necessità di interconnessione tra registri

- L'uso dei registri è quello di mantenere *copie di variabili in memoria*
  - Le unità di processamento sono all'interno della CPU
  - La velocità dei circuiti nella CPU è maggiore della velocità della memoria

- È almeno necessario supportare lo *spostamento dati* tra registri
  - Se il processore non è in grado di spostare dati, non può eseguire operazioni del tipo:

```
x = y;
```

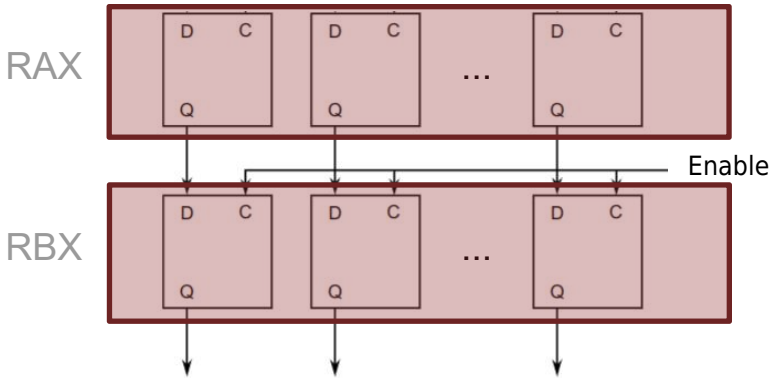
- L'istruzione di movimento dati dello z64 (nella sintassi AT&T) è:

```
MOV <sorgente>, <destinazione>
```

- Per supportare l'esecuzione di questa istruzione, servono fili per connettere tra loro i registri

# Interconnessione tra registri

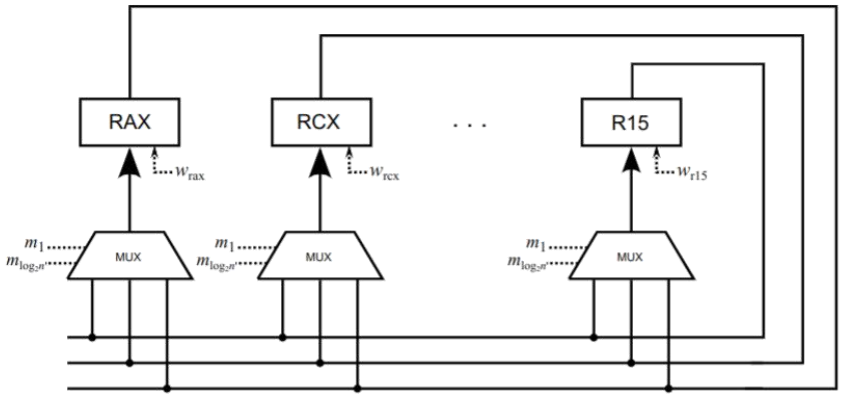
- Prima possibilità: *interconnessione diretta*
- Se vogliamo supportare l'esecuzione di `mov %rax, %rbx`, possiamo prevedere la seguente interconnessione:




- Vantaggio: semplice da progettare
- Svantaggio: complessa se occorre interconnettere più registri tra loro

# Interconnessione tra registri

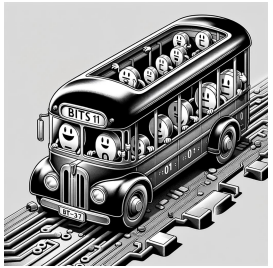
- Seconda possibilità: *interconnessione tramite multiplexer*




- Vantaggi:
  - semplice da implementare
  - possibilità di trasferire più dati contemporaneamente
- Svantaggi:
  - costo eccessivo dei multiplexer e delle linee di interconnessione
  - ancora tanti segnali di controllo: $n(\log n + 1)$

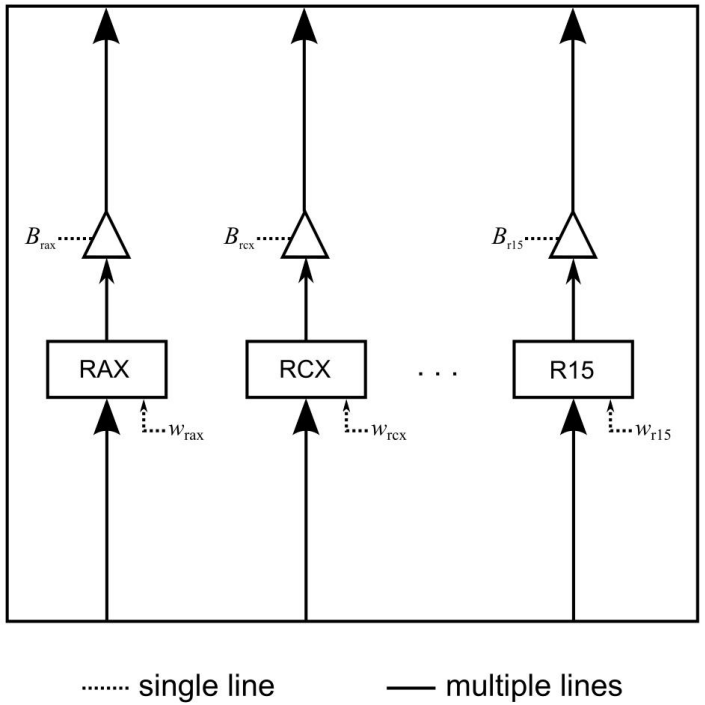
# Interconnessione tra registri

- Terza possibilità: *costruzione di un BUS* interno
- È un gruppo di 64 fili che corrono all'interno del processore
  - collega tra loro *tutti* i registri interni
- Occorre "smistare" i dati sul BUS:
  - *recupero dei dati*: i bit presenti sul BUS sono memorizzati in un registro
  - *immissione di dati*: il contenuto di un registro è posto sul BUS
- Utilizzo di più segnali di controllo opportunamente generati dalla CU:
  - Write enable per abilitare la scrittura
  - Buffer Three-State per abilitare la lettura



# Interconnessione tramite BUS interno

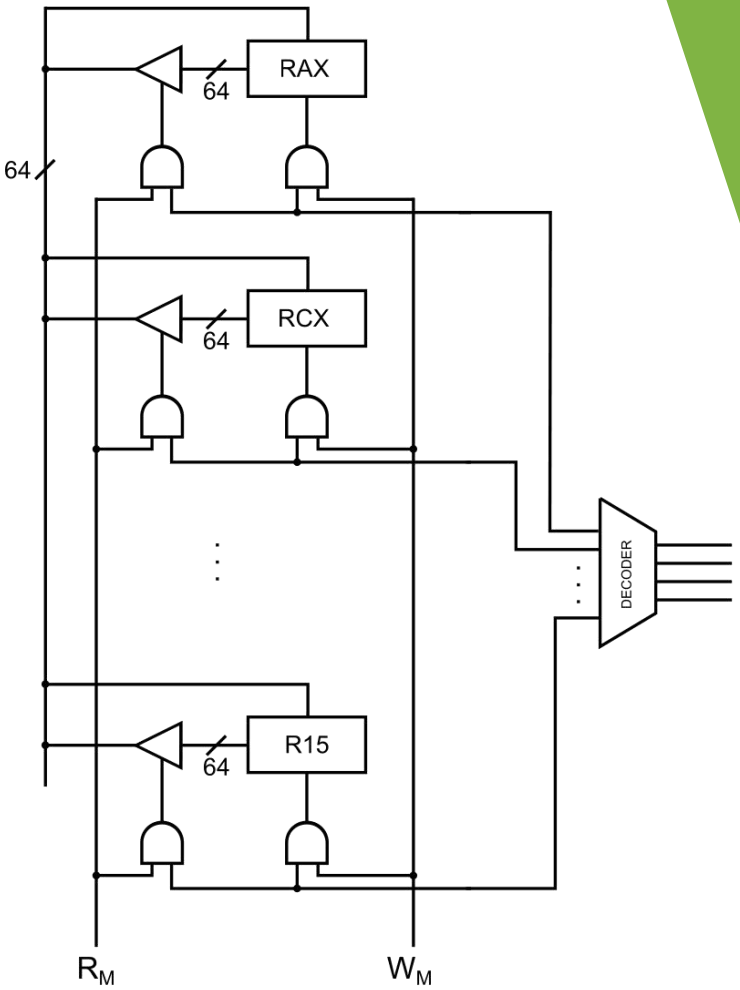
- Si può avere una sola immissione per volta sul BUS (*risorsa condivisa*)
- Pertanto, se si hanno $n$ registri, si dovranno prevedere $2n$ segnali di controllo:
  - $W_i = 1$: i dati che viaggiano sul BUS possono essere scritti nel registro $i$-esimo
  - $B_i = 1$: i dati memorizzati nel registro $i$-esimo possono essere fatti transitare sul BUS




- Vantaggio: riduzione drastica del numero di segnali di controllo necessari

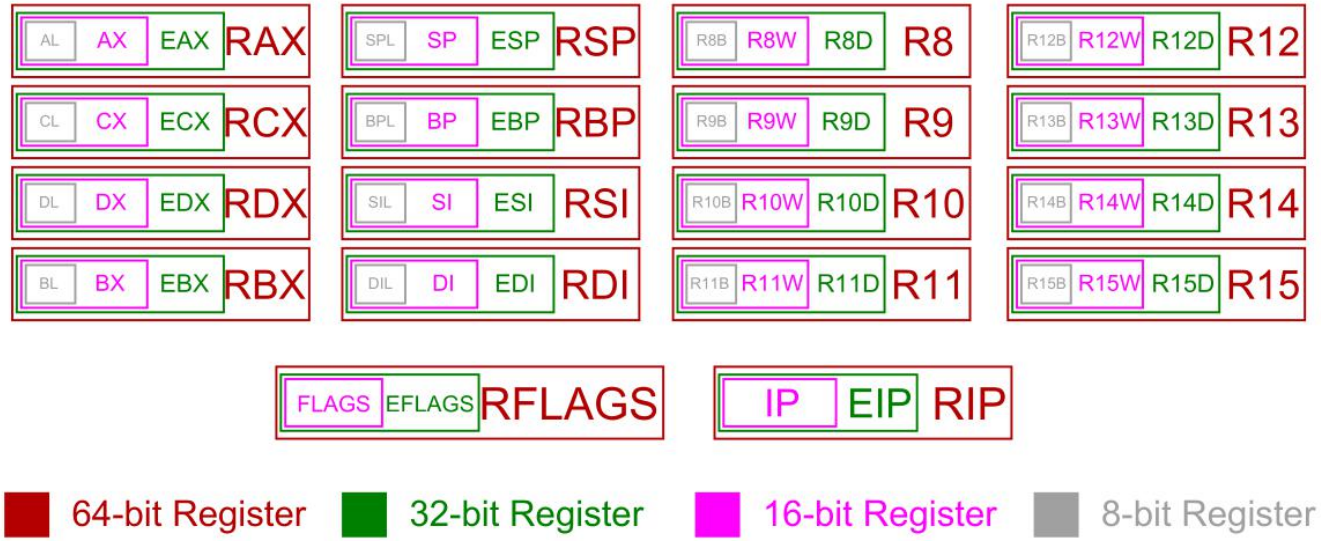
# Ottimizzazione: banco dei registri

- Ottimizzazione: *banco dei registri* (o *file dei registri* o *memoria dei registri*)
- I registri sono *codificati* con un codice numerico identificativo ∈ [0,15]
- Un decoder associa il codice di registro ad una *linea di abilitazione* del registro
- Tutti i registri sono controllati *globalmente* da due segnali di controllo:
  - $W_M$: scrittura del registro selezionato
  - $R_M$: lettura del registro selezionato
- Da dove proviene il codice al decoder?



# Processamento di dati a dimensione differente

- Un processore può operare su dati di *dimensione differente*
  - Esempi tipici: 8 bit, 16 bit, 32 bit, 64 bit
- Non è vantaggioso e pratico avere più banchi di registri
- È utile poter accedere a sottoporzioni dei dati nei registri

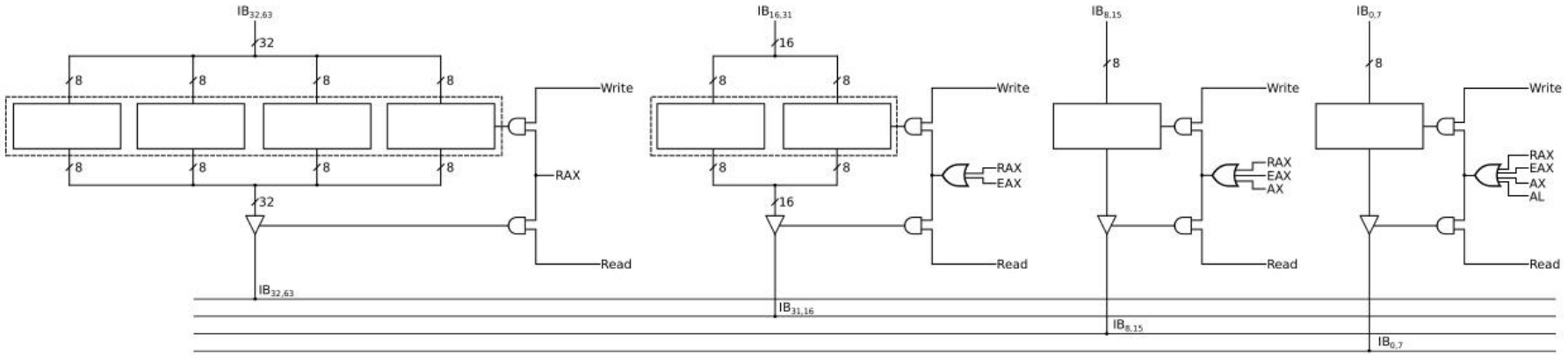


# Registri virtuali

- Le istruzioni assembly usano un suffisso per indicare la dimensione:

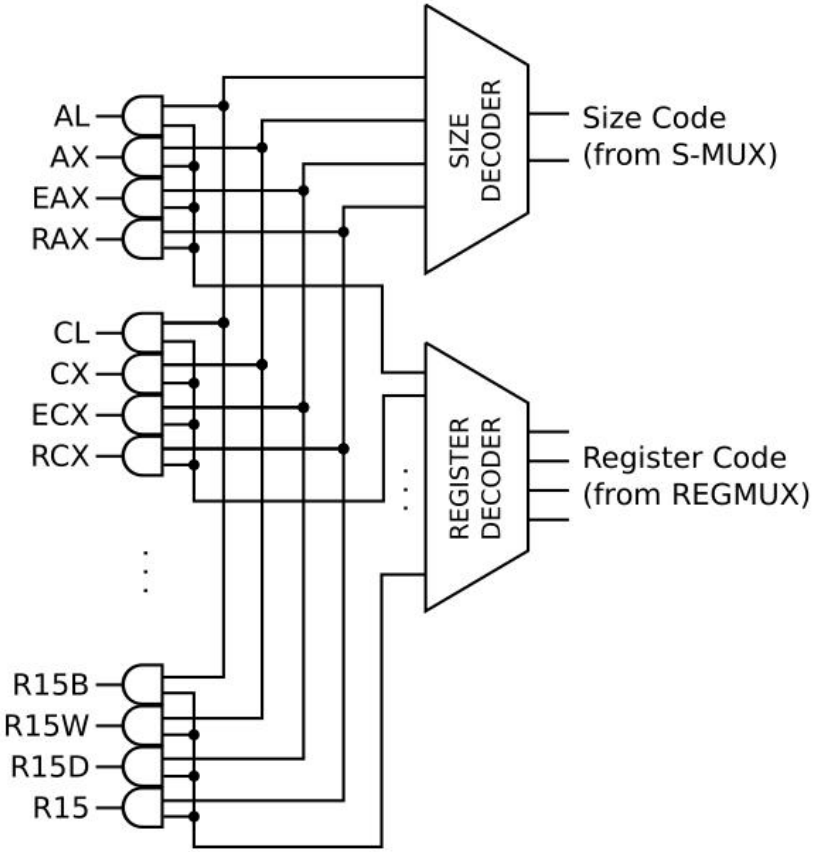
  `MOVx <sorgente>, <destinazione>`

- `B`: byte (8 bit), `W`: word (16 bit), `L`: longword (32 bit), `Q`: quadword (64 bit)
- Il suffisso fornisce parte del *contesto* alla CU per poter trattare i dati correttamente



# Registri virtuali

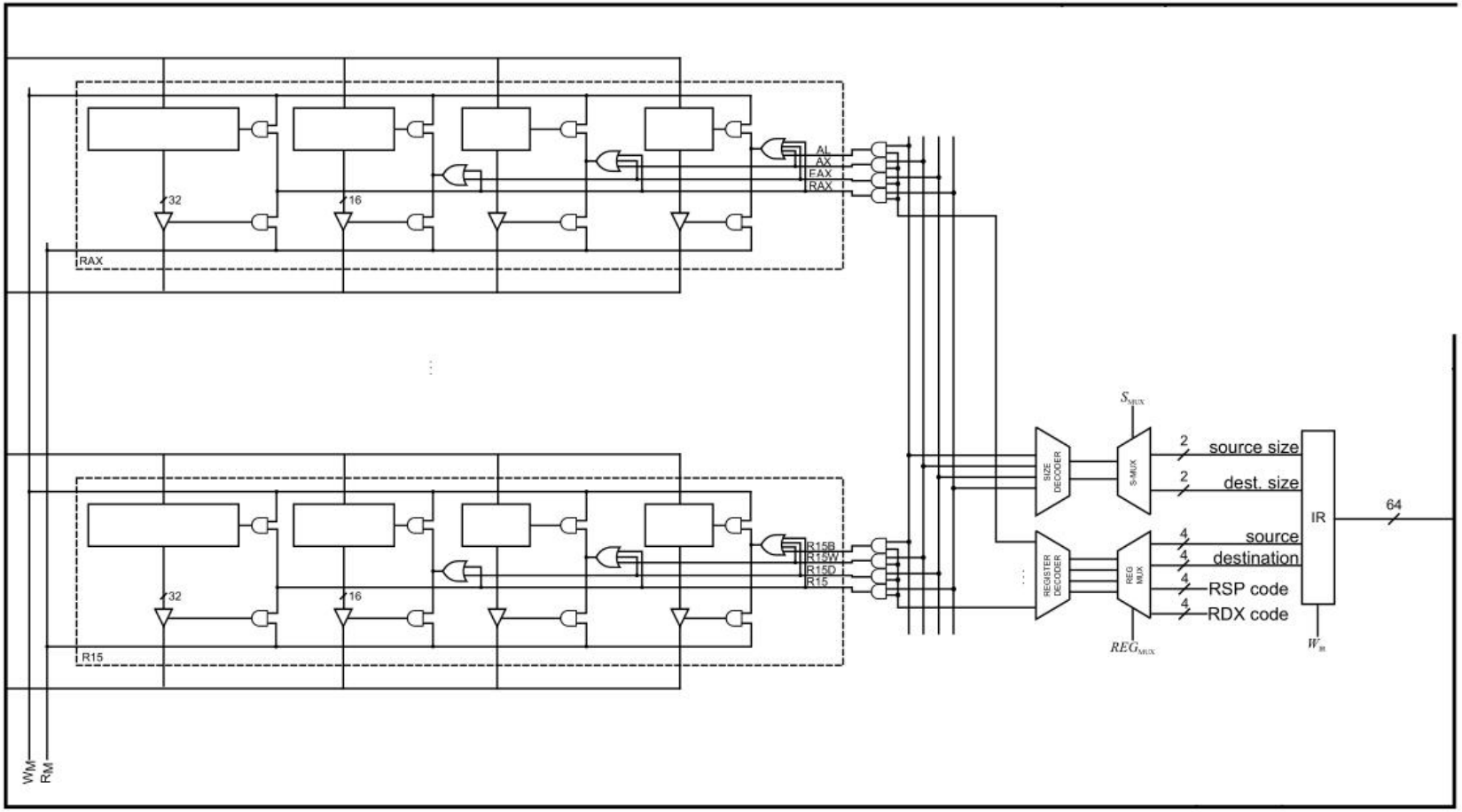
- La scelta del registro virtuale da utilizzare dipende dal suffisso dell'istruzione
- Questo è mappato sul campo SS/DS dell'istruzione
- Tali bit possono essere usati in un circuito combinatorio per realizzare un *selettore* di registro virtuale
- Di nuovo: da dove provengono i codici ai decoder?



# Instruction Register

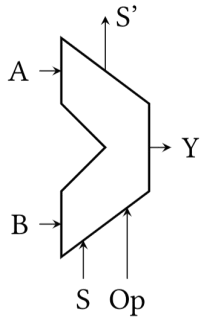
- Accedere alla memoria è un'operazione costosa
  - I registri dei processori sono realizzati con tecnologia molto più veloce, ma più costosa
  - La memoria, avendo dimensione più grande, è realizzata con tecnologie più economiche ma meno veloci
- I circuiti combinatori e sequenziali interni alla CPU devono avere gli input stabili fino alla loro stabilizzazione
- Non è ragionevole mantenere stabili gli input direttamente dalla RAM
- Si introduce un registro tampone (fondamentale): l'*Instruction Register*
- Esso mantiene una *copia* della *codifica* di un'istruzione assembly prelevata dalla memoria
- In questa codifica, sono mantenuti i codici del *registro* e della *taglia* dei dati

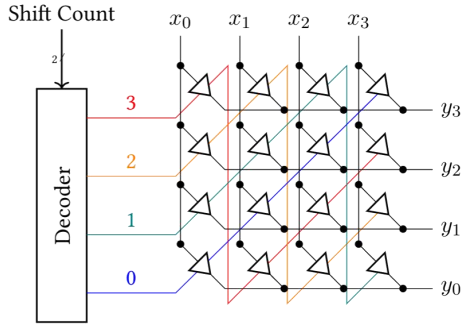
# Banco dei registri: architettura rivisitata



# Componenti per il processamento dei dati

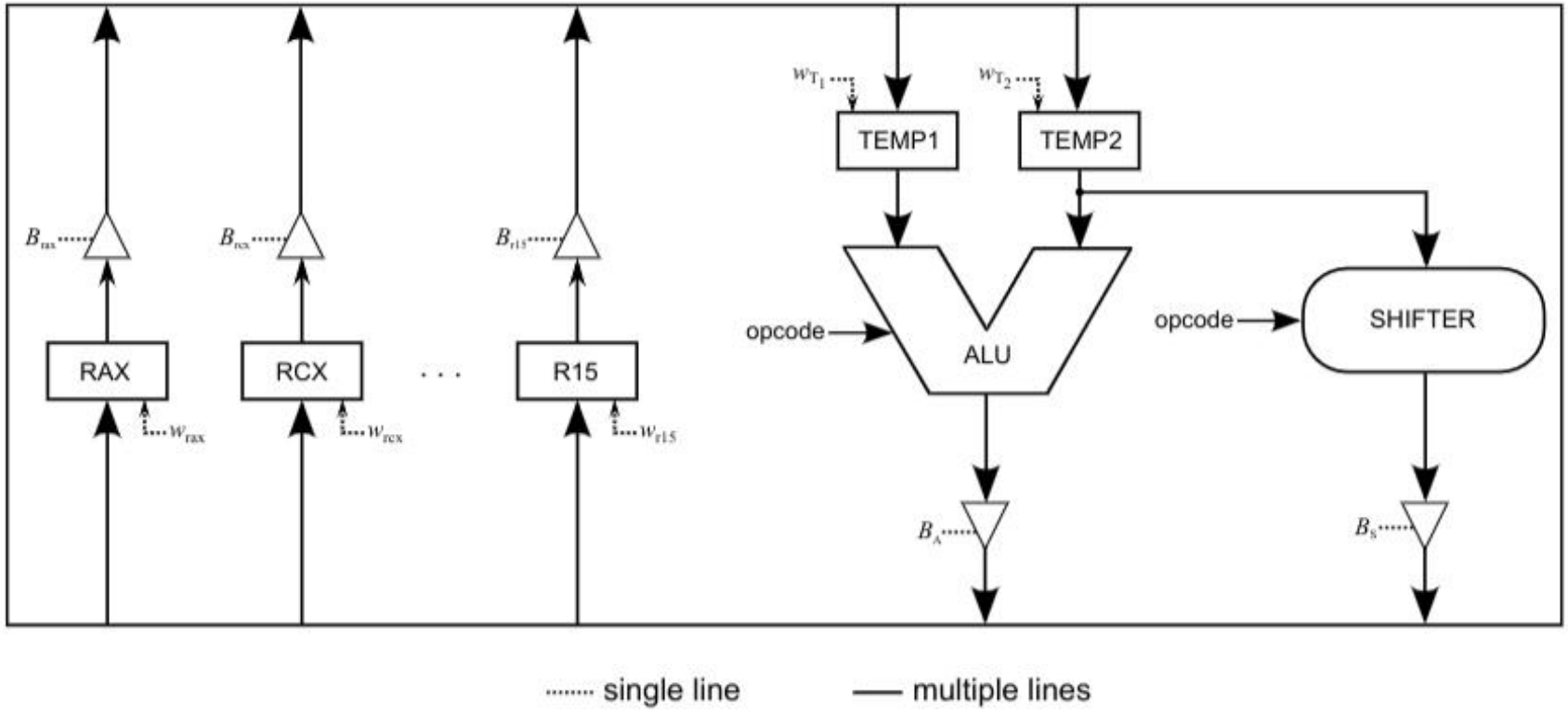
- I circuiti elementari costruiti fin'ora sono la ALU e lo shifter



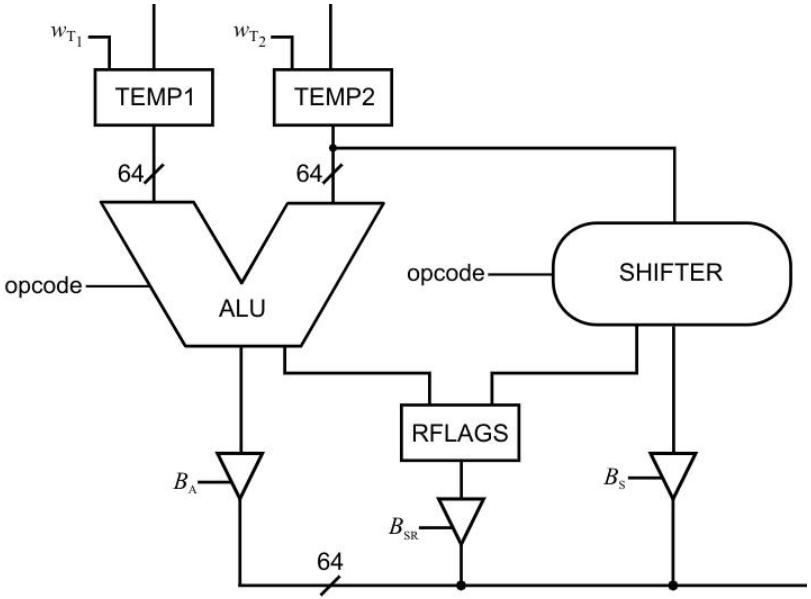

- I dati da fornire in input a questi due circuiti sono contenuti nel banco dei registri
- Problema:
  - La ALU ha bisogno di due operandi (da mantenere stabili)
  - È possibile eseguire una sola immissione di dati sul BUS interno
- Soluzione: introduzione di *registri tampone*

# Interconnessione tra registri e circuiti di calcolo



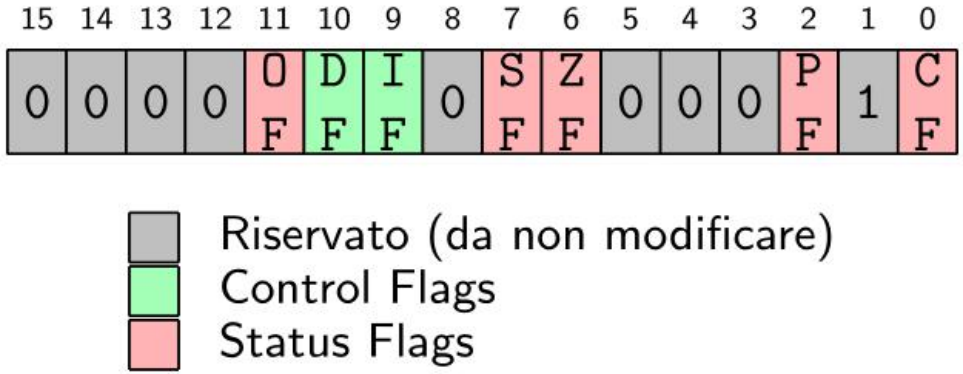
# Registro FLAGS

- ALU e Shifter sono reti iterative che emettono dei *bit di stato*
- È utile esporre lo stato al programmatore: occorre memorizzarli
- Si utilizza il registro fondamentale FLAGS



# Registro FLAGS

- I bit nel registro FLAGS si dividono in *status* e *control* bit



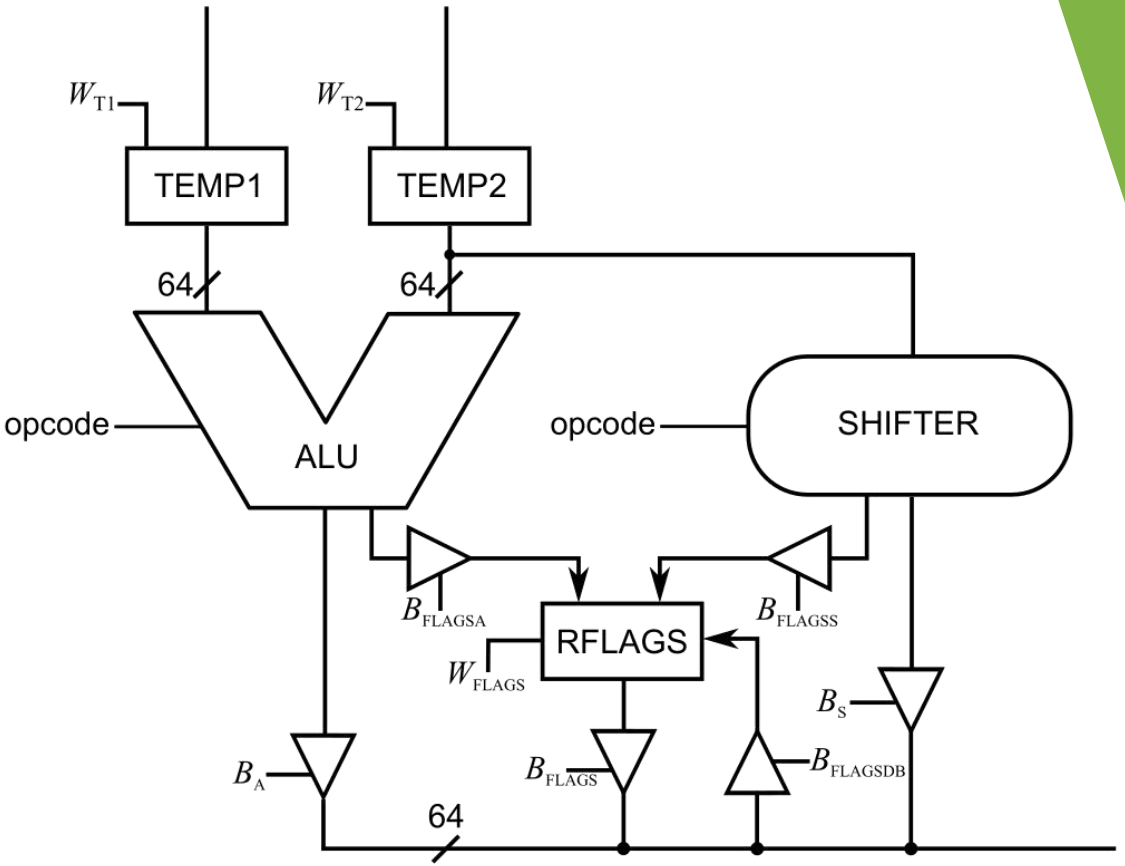

- I bit di stato sono aggiornati dalla ALU e dallo Shifter per memorizzare informazioni sull'ultima operazione eseguita
- I bit di controllo sono modificabili dal programmatore per alterare alcune funzionalità del processore

# Registro FLAGS

- **carry** (CF): vale 1 se l'ultima operazione ha prodotto un riporto
- **parity** (PF): vale 1 se nel risultato dell'ultima operazione c'è un numero pari di 1
- **zero** (ZF): vale 1 se l'ultima operazione ha come risultato 0
- **sign** (SF): vale 1 se l'ultima operazione ha prodotto un risultato negativo
- **overflow** (OF): vale 1 se il risultato dell'ultima operazione supera la capacità di rappresentazione (complemento a due)
- **interrupt enable** (IF): indica se c'è la possibilità di interrompere l'esecuzione del programma in corso
- **direction** (DF): modifica il comportamento delle operazioni su stringhe

# Memorizzare e ripristinare il registro FLAGS

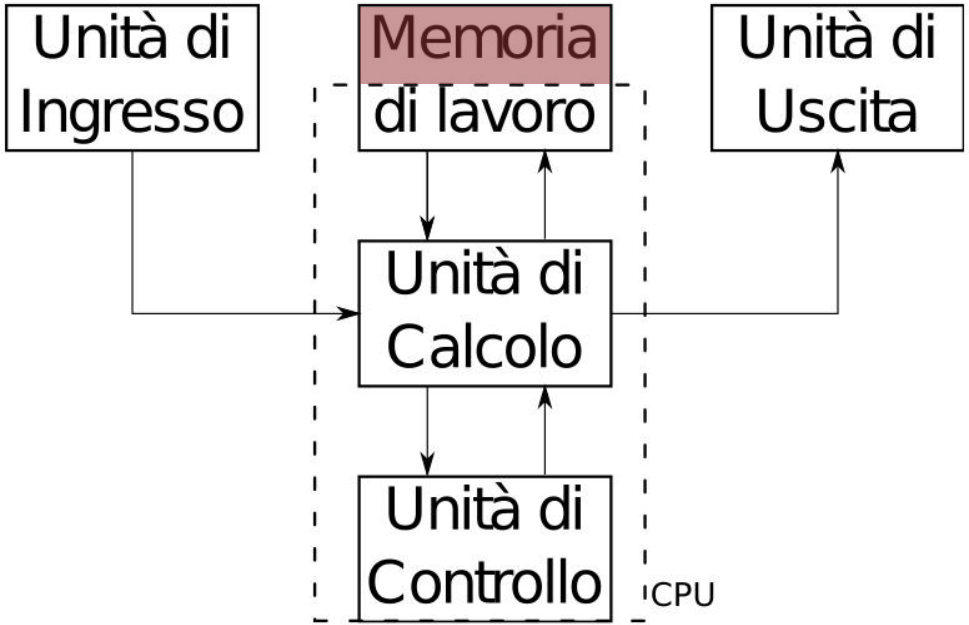
- In alcuni casi sarà necessario salvare o ripristinare il contenuto del registro FLAGS
- FLAGS è già leggibile, ma non scrivibile esplicitamente
- Occorre aggiungere un collegamento dal data BUS interno
- Tre buffer three-state determinano da dove prendere il valore di FLAGS



# Interazione con la memoria

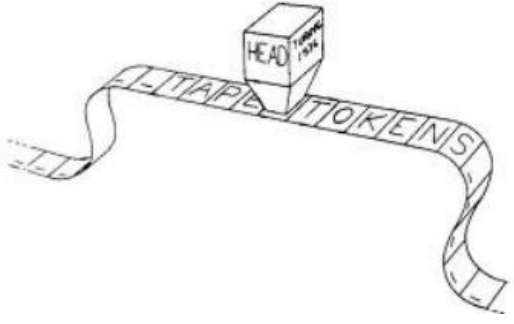
# Architettura di von Neumann rivisitata

- La restante parte della memoria di lavoro è esterna alla CPU
- È necessario prevedere un *interfacciamento* con essa ed un *protocollo* di scambio dati



# Il modello astratto di memoria di lavoro

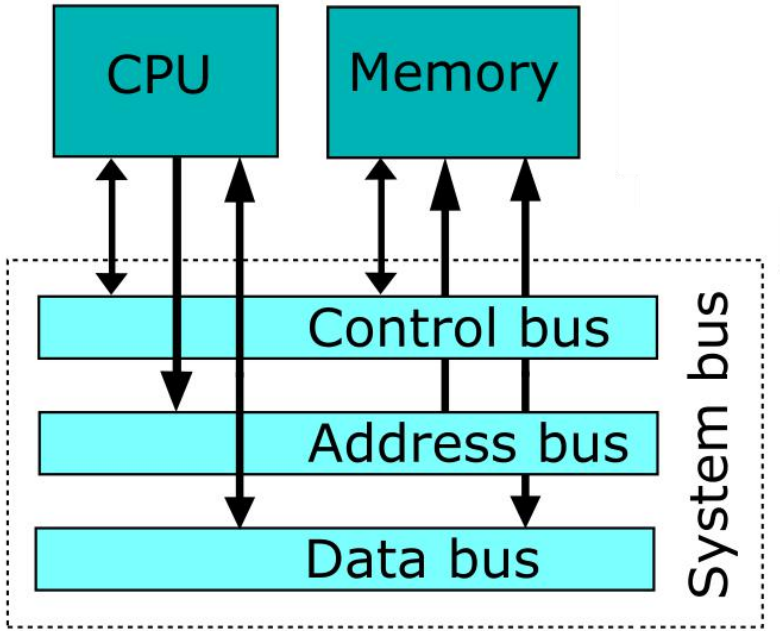
- Per la CPU, la memoria di lavoro *esterna* è un nastro molto lungo, diviso in celle (*modello di memoria piatta*)
- Ciascuna cella è identificata da un numero intero (*indirizzo*)
- Ogni cella ha una dimensione prefissata (un *byte*, tipicamente composto da 8 *bit*)
- Una "testina virtuale" si muove sul nastro per leggere o scrivere dati da/sulle celle (può essere effettuata solo una delle due operazioni alla volta)
- Una cella viene necessariamente letta o scritta nella sua interezza



# Trasferimento dati con la memoria esterna: il BUS di sistema

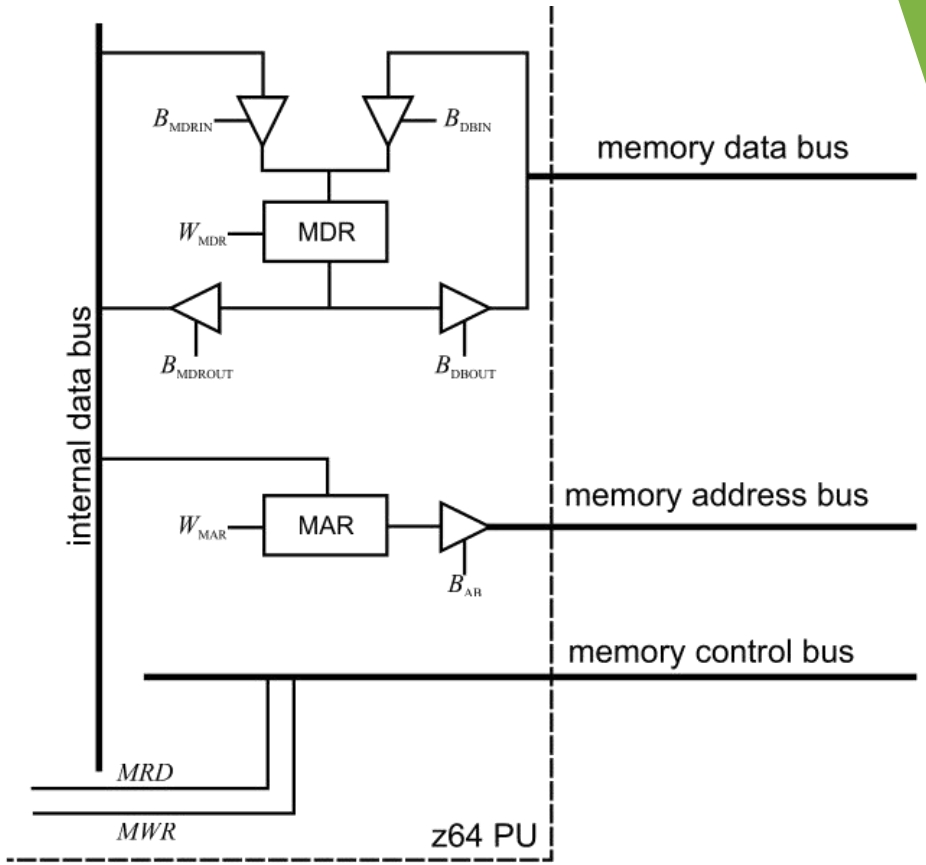
- Bus di sistema: gruppi di fili che corrono esternamente alla CPU per interconnettere la CPU con le componenti esterne (es: memoria)
- I fili sono *logicamente* suddivisi in base alla loro funzione

- Per leggere/scrivere dati, la memoria deve conoscere l'indirizzo di interesse: *address bus*
- Il processore deve poter indicare l'operazione di interesse (lettura o scrittura): *control bus*
- I dati viaggiano su fili dedicati tra CPU e memoria: *data bus*



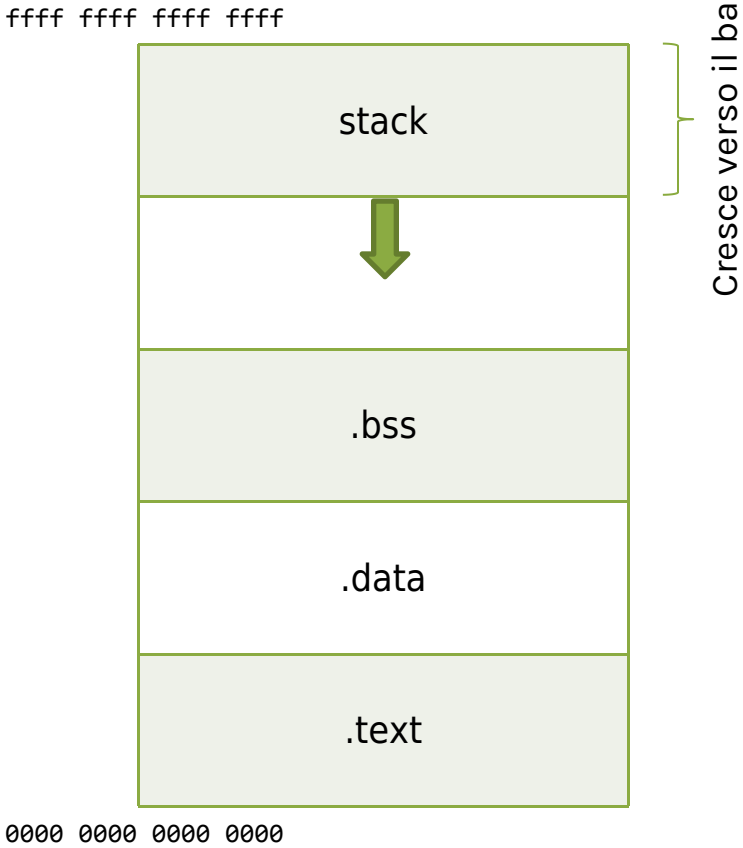
# Interfacciamento con la memoria

- Le parole ricevute dalla memoria devono essere instradate correttamente verso i registri di interesse
- I dati da scrivere in memoria devono essere letti dal registro corretto
- La memoria è tipicamente più lenta della CPU
- Occorre utilizzare un "tampone" tra i due dispositivi



# Anatomia dei programmi in memoria

- Un programma è anch'esso *codificato* in un formato binario
- Alla partenza, il programma viene caricato in memoria (von Neumann)
- `.text`: sequenza *lineare* di istruzioni (non c'è il concetto di funzione)
- `.data`/`.bss`: i dati "globali" del programma
- Ogni oggetto è identificato dal suo *indirizzo* in memoria
- Stack: la pila (architettura a stack)

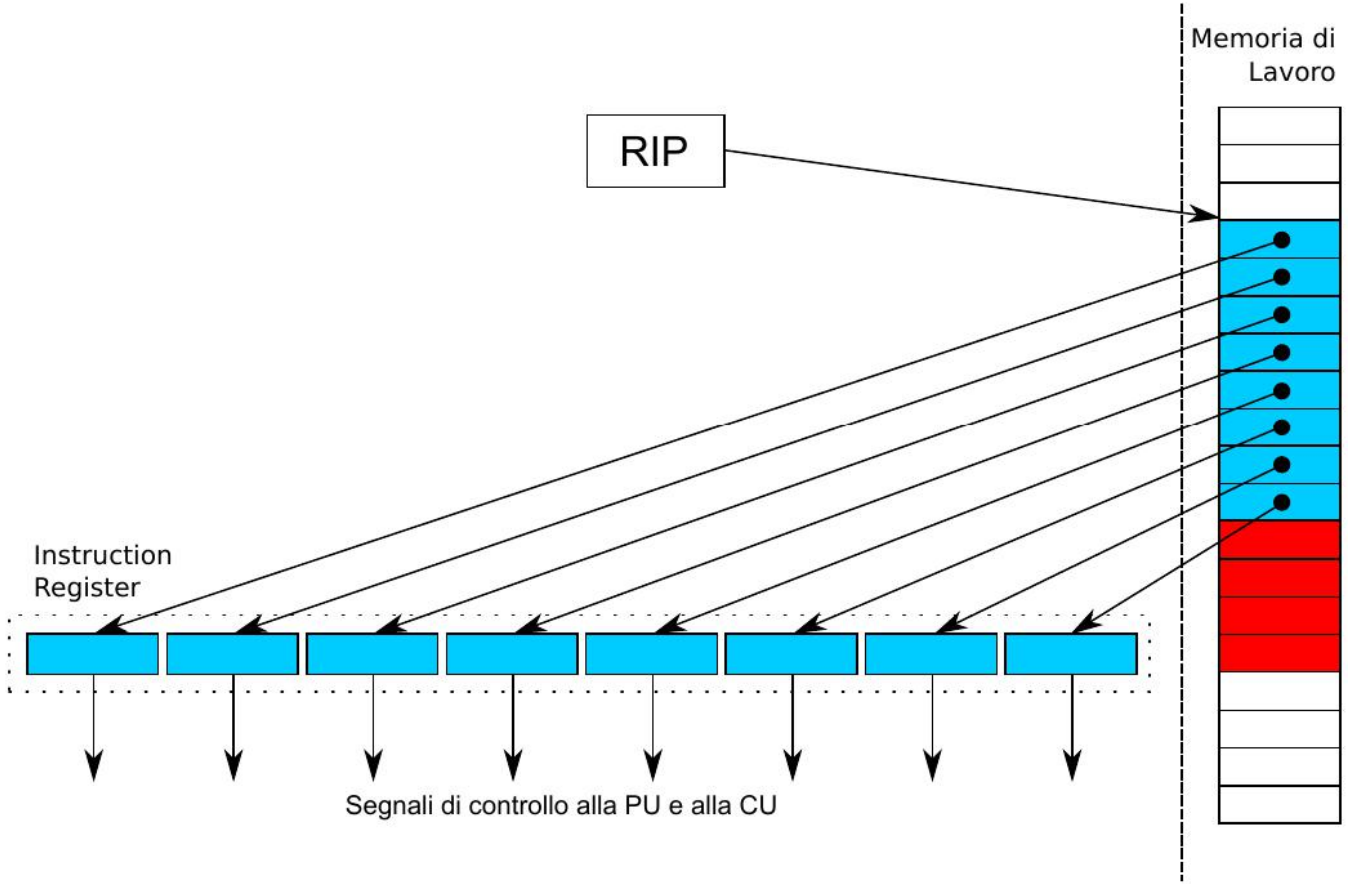


# Tracciamento dell'evoluzione del programma

- Se .text è una sequenza lineare di istruzioni, la CPU deve tenere traccia del punto in cui è arrivata ad eseguire le istruzioni
- Viene utilizzato un altro registro fondamentale: l'*instruction pointer*

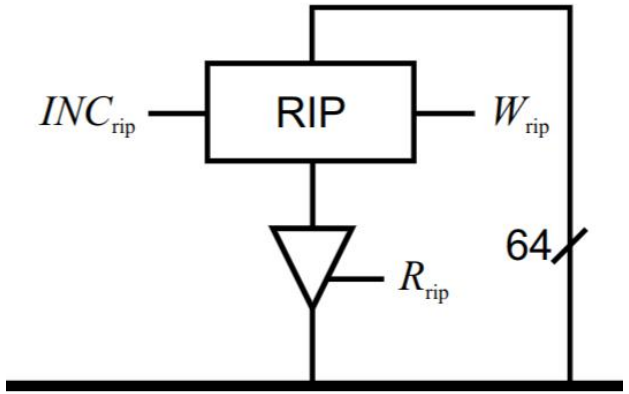
- Tale registro viene utilizzato per richiedere alla memoria di lavoro il trasferimento della *prossima istruzione da eseguire*
- RIP mantiene (quasi) sempre l'*indirizzo* della prossima istruzione (è un *puntatore a memoria*

# Prelievo di un'istruzione dalla memoria



# Incremento del registro RIP

- Il registro RIP viene incrementato dopo l'esecuzione di ogni istruzione
- Per ottimizzare le prestazioni, è possibile realizzarlo come *registro a incremento*
- Il registro è accoppiato ad un sommatore veloce dedicato
- Svantaggio: circuito più complesso
- Vantaggio: l'incremento può essere eseguito *in parallelo* all'esecuzione dell'istruzione corrente



# Lo stack di programma

- Il processore ha un numero estremamente limitato di registri
- Un programma potrebbe avere bisogno di gestire tante variabili o variabili molto grandi
- È possibile utilizzare un'area di memoria come "area di appoggio": lo stack (*pila*) di programma
- È una struttura dati di tipo *LIFO* (Last-In First-Out): il primo elemento che può essere prelevato è l'ultimo ad essere stato memorizzato
- Si possono effettuare due operazioni su questa struttura dati:
  - *push*: viene inserito un elemento sulla sommità (*top*) della pila
  - *pop*: viene prelevato l'elemento affiorante (*top element*) dalla pila
- Data l'importanza di questa struttura dati, molte ISA forniscono istruzioni dedicate per la sua manipolazione

# Gestione dello stack nello z64

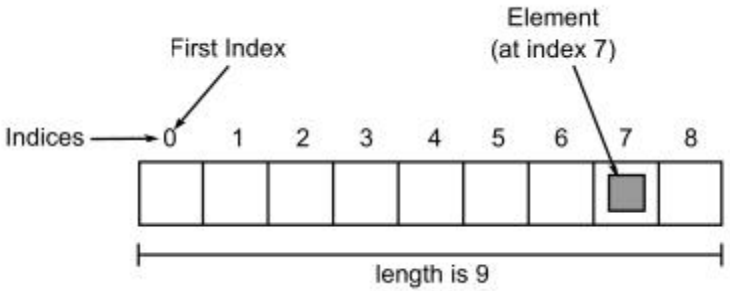
- Lo stack è composto da quadword (non si può eseguire una push di un singolo byte)
- La cima dello stack è individuata dall'indirizzo memorizzato in un registro specifico chiamato *stack pointer*
- Lo stack "cresce" se il valore contenuto in SP diminuisce, "decresce" se il valore contenuto in SP cresce
- Le istruzioni che implementano le operazioni di push e pop utilizzano il registro RSP in modo *implicito*
- Modificare *esplicitamente* il valore di RSP significa perdere il *riferimento* alla cima dello stack, e quindi a tutto il suo contenuto
  - È però accessibile al programmatore: si può decidere di utilizzare più stack
  - È quello che fanno i sistemi operativi per eseguire più processi contemporaneamente

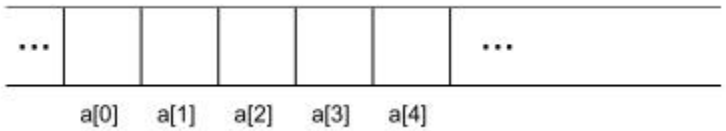
# Modalità di Indirizzamento

# Strutture dati e modello di memoria lineare

- Il modello astratto di memoria è *lineare*, quindi *indirizzato al byte*
- I linguaggi di programmazione di più alto livello offrono astrazioni differenti:
  - Vettori: `x = A[i];`
  - Strutture dati: `x = str.member;`

- Il programmatore (o il compilatore) può scrivere un programma assembly per tradurre un accesso ad un elemento di una struttura dati in un *indirizzo effettivo* di memoria

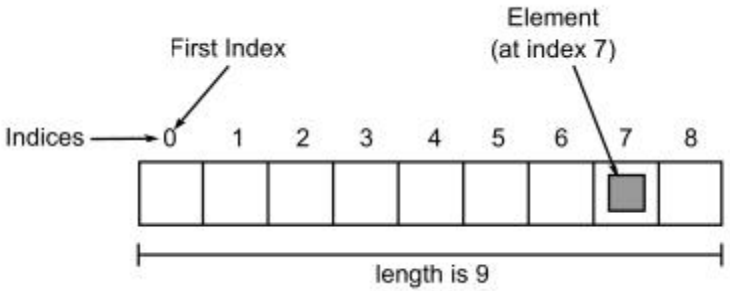


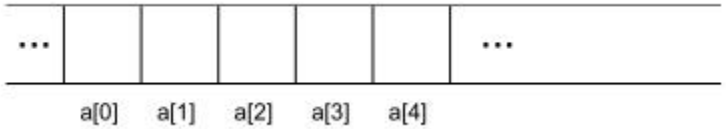



# Strutture dati e modello di memoria lineare

- Ad esempio, un accesso all'elemento *i*-esimo di un vettore può essere "tradotto" in un indirizzo effettivo applicando l'operatore *spiazzamento*.
- Se la variabile `a` è associata all'indirizzo di *base* del vettore e tutti gli elementi hanno la stessa dimensione:

$$\texttt{a[i]} \Leftrightarrow \texttt{a + size * i}$$

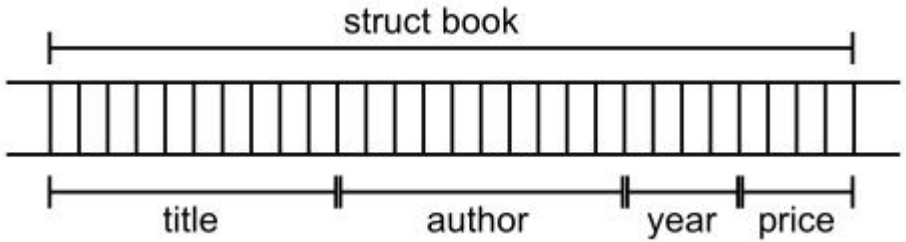


# Strutture dati e modello di memoria lineare

- Strutture dati più complesse posso utilizzare celle di memoria contigue per memorizzare dati di tipo differente
- La memoria è lineare: occorre dare un *contesto* a ciascun *membro* della struttura

```
struct book {
    char title[10];
    char author[10];
    int publication_year;
    float price;
};
```




- Se una variabile di tipo struttura `libro` è associata all'indirizzo di *base* in memoria in cui si trova la struttura, accedere a un membro equivale a calcolare:

`libro.year` ⟺ `libro + spiazzamento(titolo) + spiazzamento(autore)`

# Modalità di indirizzamento in memoria

- Poiché stiamo realizzando un'architettura CISC, è sensato offrire un supporto migliore al calcolo degli *indirizzi effettivi* a livello di CPU
- Nei casi (molto comuni) di vettori e strutture, gli elementi ricorrenti per il calcolo di un indirizzo effettivo sono stati:
  - indirizzo di base
  - indice
  - spiazzamento
- Per quanto riguarda l'indice dei vettori, è utile anche conoscere la taglia dei singoli elementi del vettore:

  `a[i]` ⟺ `a + size * i`
- Poiché i tipi primitivi della CPU sono a 8, 16, 32 e 64 bit, è utile prevedere una *scala* pari a questi valori

# Modalità di indirizzamento in memoria

- Le istruzioni assembly possono accettare anche *operandi in memoria*
  - Per motivi di codifica, al più uno solo
  - `MOVx <sorgente>, <destinazione>`
  - Uno tra `<sorgente>` e `<destinazione>` può essere un operando in memoria
- Per supportare le operazioni comuni, gli operandi in memoria hanno la forma:

  `spiazzamento(base, indice, scala)`

- Tutte le componenti sono *opzionali* (ma se c'è l'indice, c'è anche la scala)
- Base e indice sono *registri general purpose*
  - base: si può riusare lo stesso codice per accedere, ad esempio, a vettori differenti
  - indice: si può utilizzare la stessa istruzione all'interno di un ciclo per scandire un vettore
- Scala e spiazzamento sono delle *costanti* codificate nell'istruzione

# Modalità di indirizzamento in memoria

$$\left[\left\{\begin{array}{c}\text{AX}\\ \text{BX}\\ \text{CX}\\ \text{DX}\\ \text{SP}\\ \text{BP}\\ \text{SI}\\ \text{DI}\\ \text{R8}\\ \text{R9}\\ \text{R10}\\ \text{R11}\\ \text{R12}\\ \text{R13}\\ \text{R14}\\ \text{R15}\end{array}\right\}\right] + \left[\left\{\begin{array}{c}\text{AX}\\ \text{BX}\\ \text{CX}\\ \text{DX}\\ \text{SP}\\ \text{BP}\\ \text{SI}\\ \text{DI}\\ \text{R8}\\ \text{R9}\\ \text{R10}\\ \text{R11}\\ \text{R12}\\ \text{R13}\\ \text{R14}\\ \text{R15}\end{array}\right\} * \left\{\begin{array}{c}1\\ 2\\ 4\\ 8\end{array}\right\}\right] + [\textit{spiazzamento}]$$

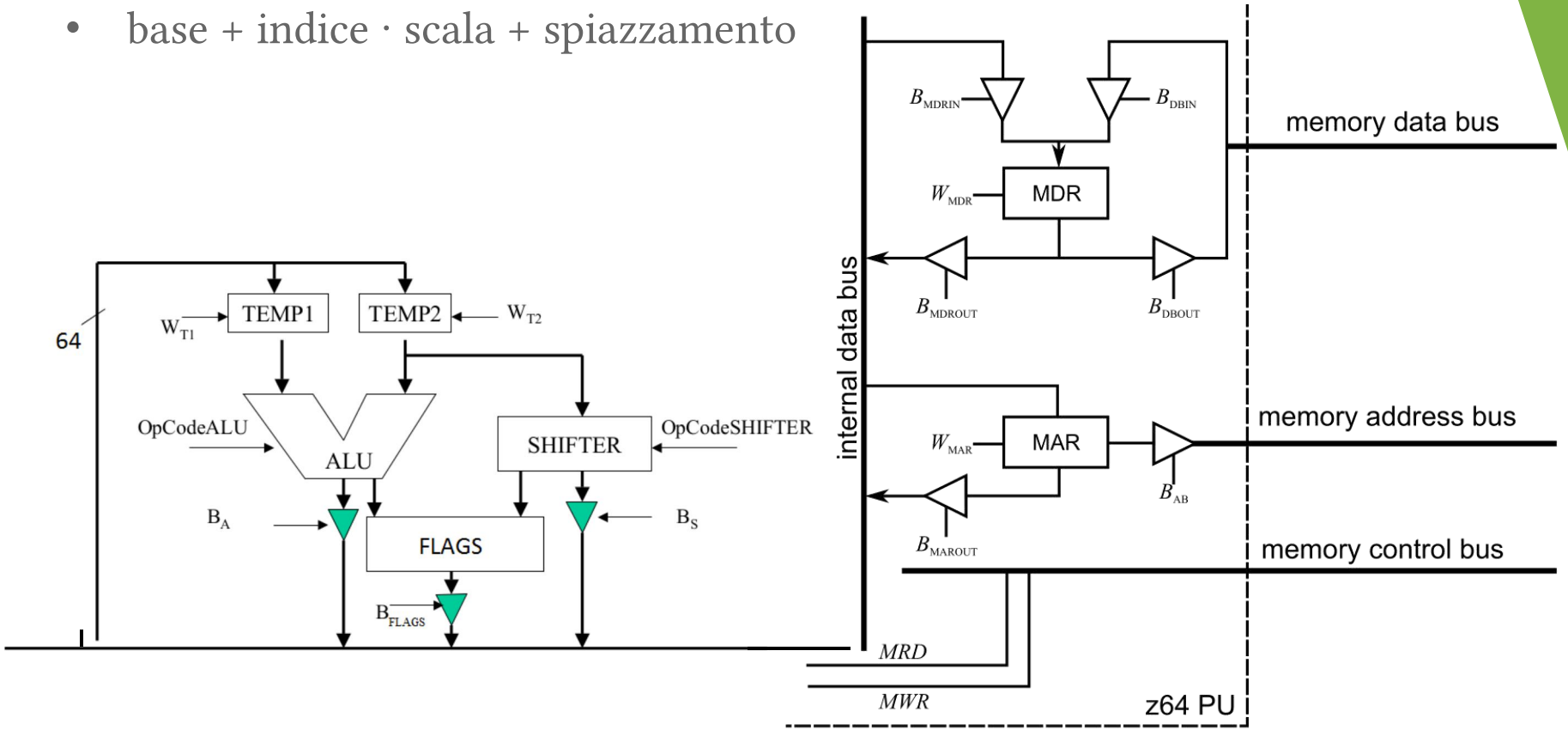
# Implementazione della modalità di indirizzamento

- La determinazione di un indirizzo di memoria è un'operazione complessa
- È necessario fare affidamento sulla PU per calcolare questo indirizzo
- Due possibilità:
  - Introduzione di hardware dedicato
  - Utilizzo dell'hardware già presente

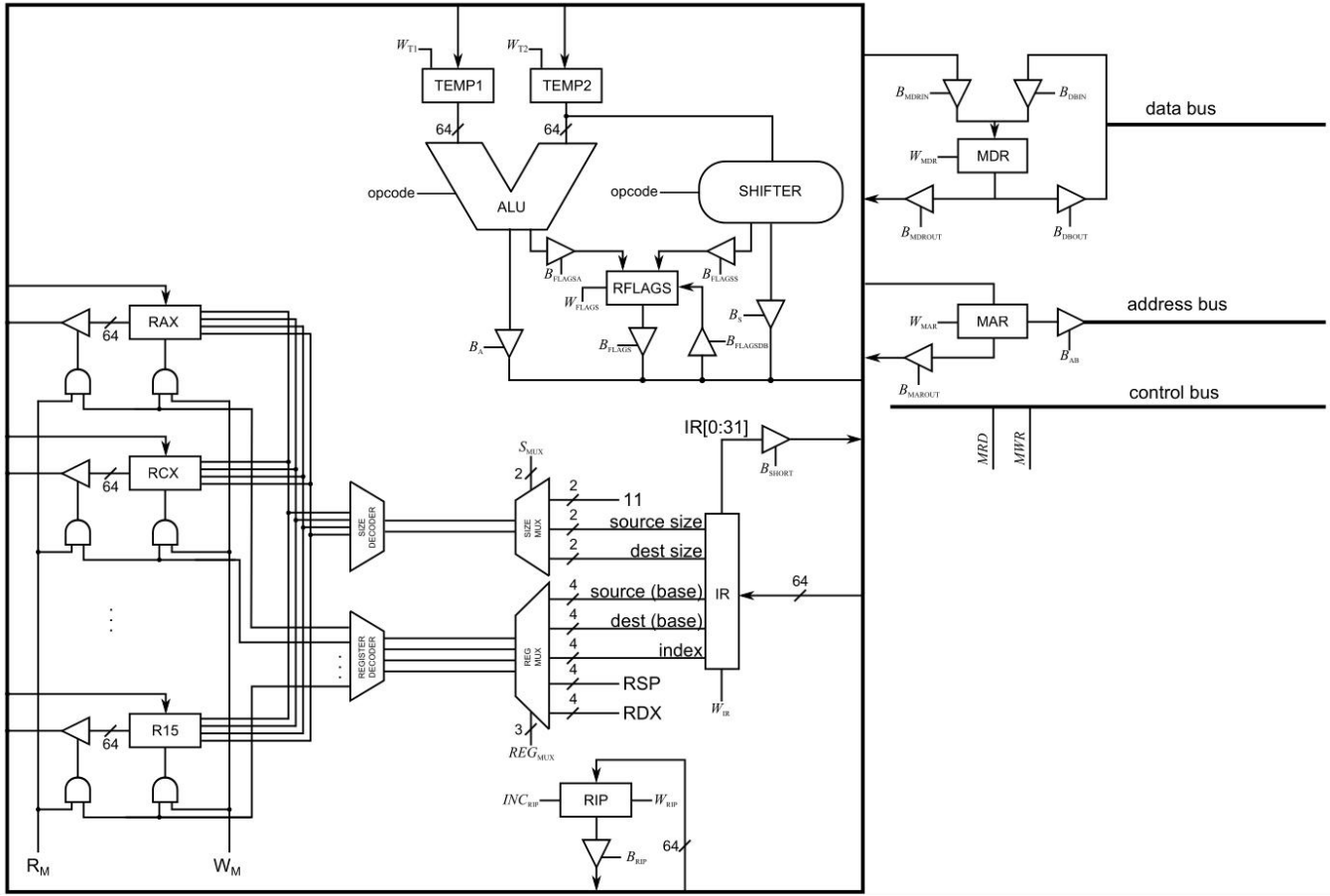
- Qual è la soluzione più conveniente?

# Lettura del registro MAR

- base + indice · scala + spiazzamento



# Architettura finale della PU



# Formato Istruzioni

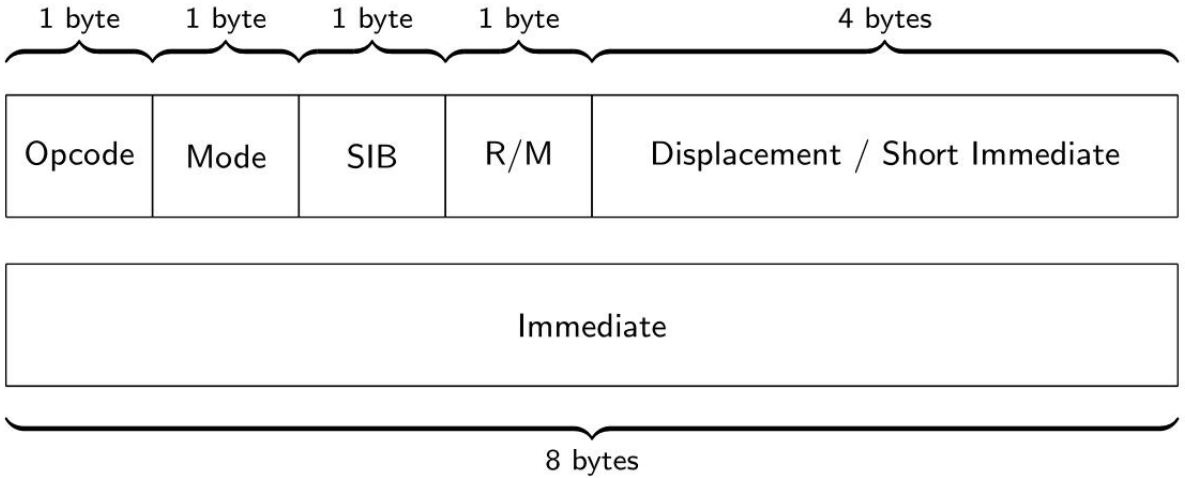
## Come codificare le istruzioni assembly in un formato binario

# Istruzioni dello z64

- Definire una *codifica* delle istruzioni è fondamentale per progettare un'ISA
  - Determina in che modo la CU interpreta le istruzioni
  - Determina quali sequenze di parole binarie il compilatore inserisce in `.text`
- Sono organizzate in otto classi:
  - Classe 0: Controllo dell'hardware
  - Classe 1: Spostamento dati
  - Classe 2: Aritmetiche (su interi) e logiche
  - Classe 3: Rotazione e shift
  - Classe 4: Operazioni sui bit di FLAGS
  - Classe 5: Controllo del flusso d'esecuzione del programma
  - Classe 6: Controllo condizionale del flusso d'esecuzione del programma
  - Classe 7: Ingresso/uscita di dati

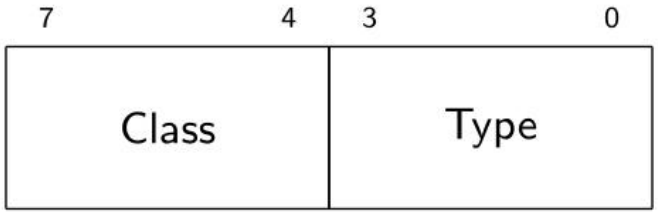
# Formato istruzioni macchina

- Le istruzioni hanno un formato a *lunghezza variabile*



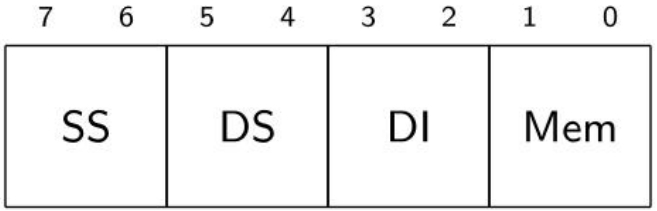

- La lunghezza variabile permette la generazione di programmi più compatti (meno consumo di memoria e spazio su disco)

# Il campo Opcode




- Class è un codice di 4 bit che identifica la famiglia di istruzioni cui appartiene quella corrente
- Type è un codice di quattro bit che identifica la precisa istruzione nella famiglia
- Questa differenziazione ci consentirà di realizzare una CU più ottimizzata

# Il campo Mode

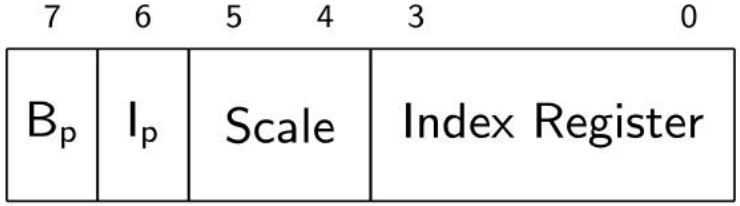



- SS e DS contengono rispettivamente la codifica della dimensione dell'operando sorgente e destinazione
- DI è un campo di 2 bit che indica o meno la presenza di un displacement e di un dato immediato
- Mem indica quali degli operandi (sorgente o destinazione) sono da considerarsi operandi in memoria

# Il campo Mode

| Campo | Valore | Significato |
|---|---|---|
| SS | 00 | La sorgente è un byte |
| | 01 | La sorgente è una word |
| | 10 | La sorgente è una longword |
| | 11 | La sorgente è una quadword |
| DS | 00 | La destinazione è un byte |
| | 01 | La destinazione è una word |
| | 10 | La destinazione è una longword |
| | 11 | La destinazione è una quadword |
| DI | 00 | Spiazzamento non utilizzato, immediato non presente |
| | 01 | Immediato presente |
| | 10 | Spiazzamento utilizzato |
| | 11 | Spiazzamento utilizzato, immediato presente |
| Mem | 00 | Sia la sorgente che la destinazione sono registri |
| | 01 | La sorgente è un registro, la destinazione è in memoria |
| | 10 | La sorgente è in memoria, la destinazione è un registro |
| | 11 | Condizione impossibile (genera un'eccezione a runtime) |

# Il campo SIB (Scala, Indice, Base)



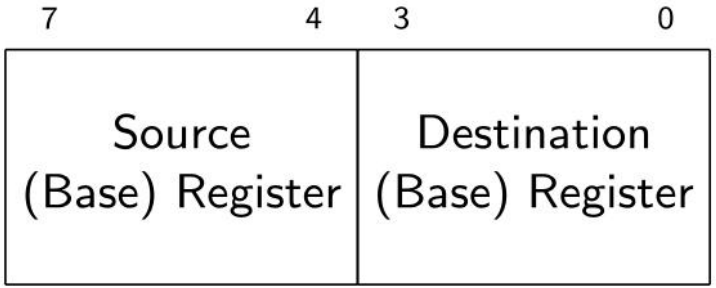

- $B_p$ e $I_p$ indicano se si deve utilizzare una base e/o un indice per calcolare l'indirizzo effettivo
- Se $I_p$==1, il campo Index mantiene la codifica binaria del registro indice
- Scale mantiene il valore della scala, i cui valori leciti sono 1 (codificato come 00), 2 (codificato come 01), 4 (codificato come 10) e 8 (codificato come 11)

# Codifica dei registri fisici

| Nome | Codifica | Uso Comune |
|---|---|---|
| RAX | 0000 | Registro Accumulatore |
| RCX | 0001 | Registro Contatore |
| RDX | 0010 | Registro Dati |
| RBX | 0011 | Registro Base |
| RSP | 0100 | Stack Pointer |
| RBP | 0101 | Base Pointer |
| RSI | 0110 | Registro Sorgente |
| RDI | 0111 | Registro Destinazione |
| R8 | 1000 | Registro di uso generale |
| R9 | 1001 | Registro di uso generale |
| R10 | 1010 | Registro di uso generale |
| R11 | 1011 | Registro di uso generale |
| R12 | 1100 | Registro di uso generale |
| R13 | 1101 | Registro di uso generale |
| R14 | 1110 | Registro di uso generale |
| R15 | 1111 | Registro di uso generale |

- Alcune istruzioni utilizzando degli *operandi impliciti*
- Alcuni registri hanno un ruolo particolare che queste istruzioni sfruttano
- Se si utilizzano istruzioni senza operandi impliciti, i registri possono essere usati come *general purpose* indicandoli come *operandi espliciti*

# Il campo R/M




- Questo campo ha spazio per mantenere esattamente due codifiche di registri
- L'interpretazione di questi campi dipende dai valori dei sottocampi di Mem e del bit $B_p$ di SIB
- I registri possono quindi essere interpretati come registri semplici oppure come registri base

# Alcune istruzioni assembly

- `movb %al, %bl`: copia il contenuto del byte meno significativo di RAX in RBX
- `movw %ax, (%rdi)`: copia il contenuto dei 2 byte meno significativi di RAX nei due byte di memoria il cui indirizzo iniziale è memorizzato in RDI
- `movl (%rsi), %eax`: copia nei 4 byte meno significativi di RAX il contenuto dei 4 byte di memoria il cui indirizzo è specificato in RSI.

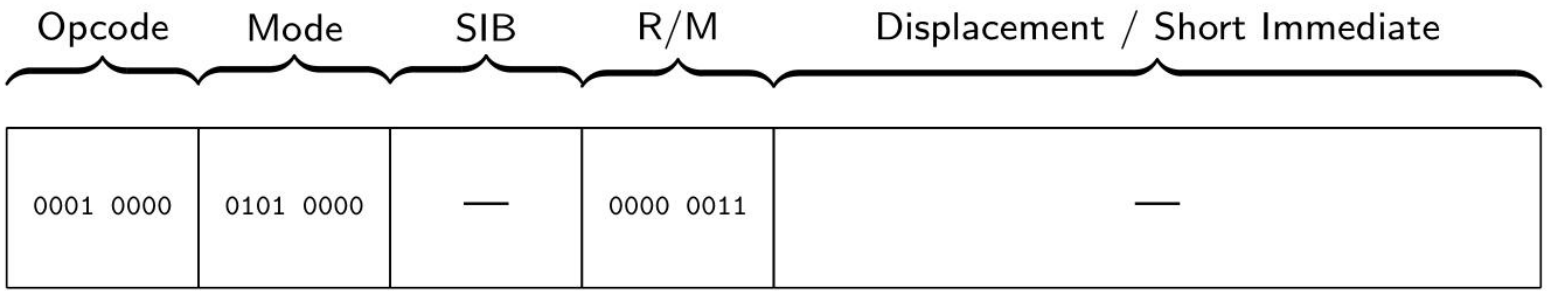
# Alcune istruzioni assembly

- `movq (%rsi, %rcx, 8), %rax`: copia nel registro RAX il contenuto degli 8 byte di memoria il cui indirizzo iniziale è calcolato come RSI + RCX · 8
- `subl %eax, %edx`: sottrai il contenuto dei 4 byte meno significativi di RAX dai 4 byte meno significativi di RDX e aggiorna il contenuto dei 4 byte meno significativi di RDX
- `addb $d, %al`: somma al byte meno significativo di RAX la quantità costante `d`.
- `addb d, %al`: somma al byte meno significativo di RAX il byte contenuto all'indirizzo di memoria `d`.

# Traduzione istruzioni assembly in linguaggio macchina

```
movw %ax, %bx
```

- Non si ha alcun accesso in memoria
- Sorgente e destinazione sono entrambi di 2 byte
- Non è coinvolta alcuna costante

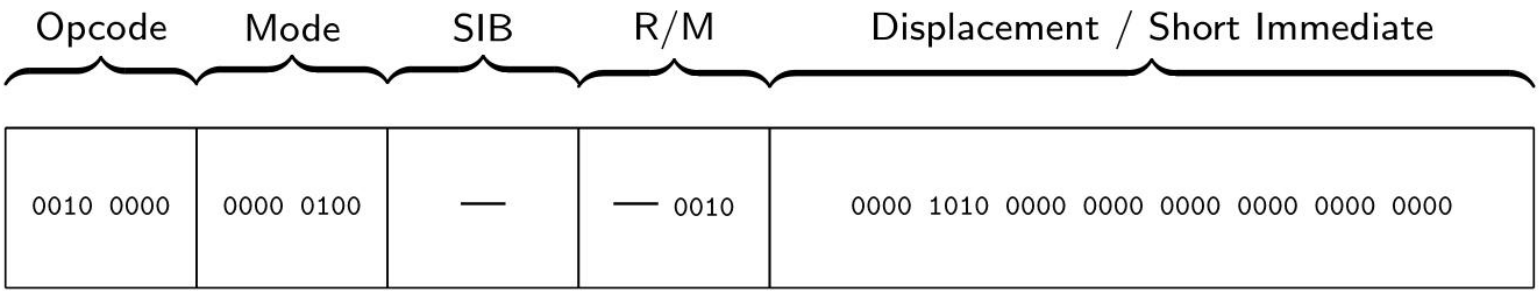



```
10 50 00 03 00 00 00 00
```

# Traduzione istruzioni assembly in linguaggio macchina

```
addb $0xa, %dl
```

- Si utilizza una costante numerica come operando
- Non è presente uno spiazzamento

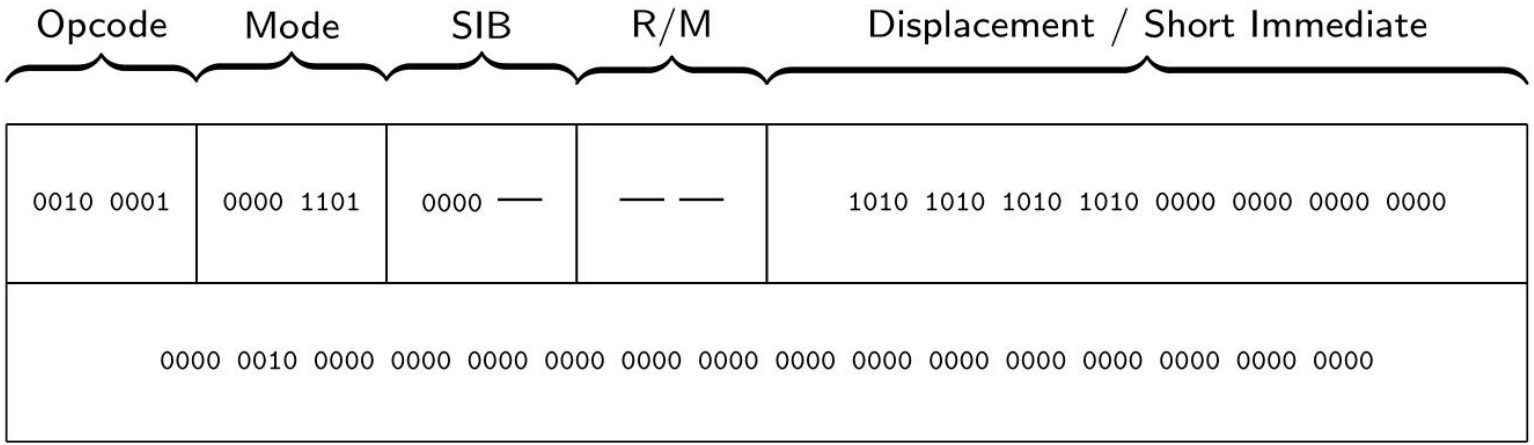



```
20 04 00 02 0A 00 00 00
```

# Traduzione istruzioni assembly in linguaggio macchina

```
subb $0x2, 0xAAAA
```

- Si utilizza una costante numerica come operando
- È presente uno spiazzamento

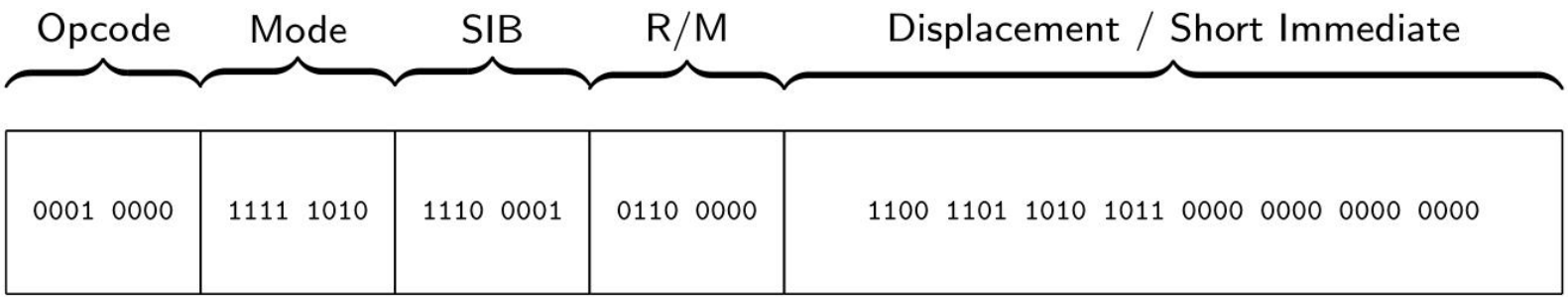



```
21 0D 00 00 AA AA 00 00 02 00 00 00 00 00 00 00
```

# Traduzione istruzioni assembly in linguaggio macchina

```
movq 0xabcd(%rsi, %rcx, 4), %rax
```

- La sorgente è un operando in memoria
- Per accedere in memoria vengono usati scala, indice, base e spiazzamento




```
10 F8 E1 60 CD AB 00 00
```

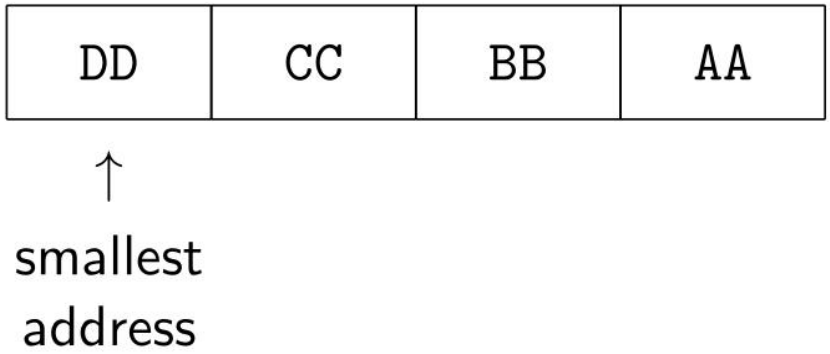
# Memory Endianness: Ordine dei byte

- L'ordine dei byte descrive la modalità utilizzata dai calcolatori per immagazzinare in memoria dati di dimensione superiore al byte
- La differenza dei sistemi è data dall'ordine con il quale i byte costituenti il dato da immagazzinare vengono memorizzati:
  - *litte-endian*: memorizzazione che inizia dal byte meno significativo per finire col più significativo (usato dai processori Intel)
  - *big-endian*: memorizzazione che inizia dal byte più significativo per finire col meno significativo (Network-byte order)
  - *middle-endian*: ordine dei byte né crescente né decrescente (es: 3412, 2143)

- La differenza si rispecchia nel *network-byte order* vs *host order*

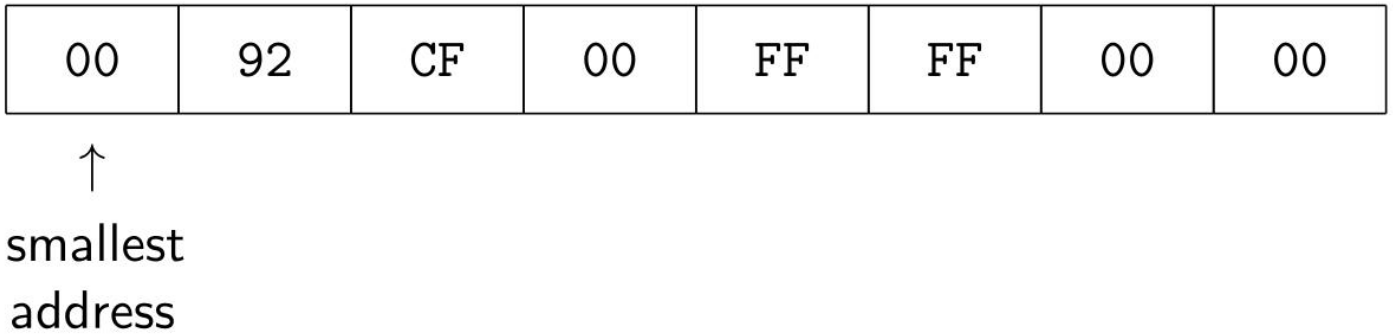
# Effetti della Little-Endianness

- Il processore z64 accede in memoria secondo lo schema little-endian
- Valori multibyte sono memorizzati con il loro byte meno significativo all'indirizzo più basso
- Il valore `0xAABBCCDD` è posto nel layout di memoria come segue:



# Effetti della Little-Endianness

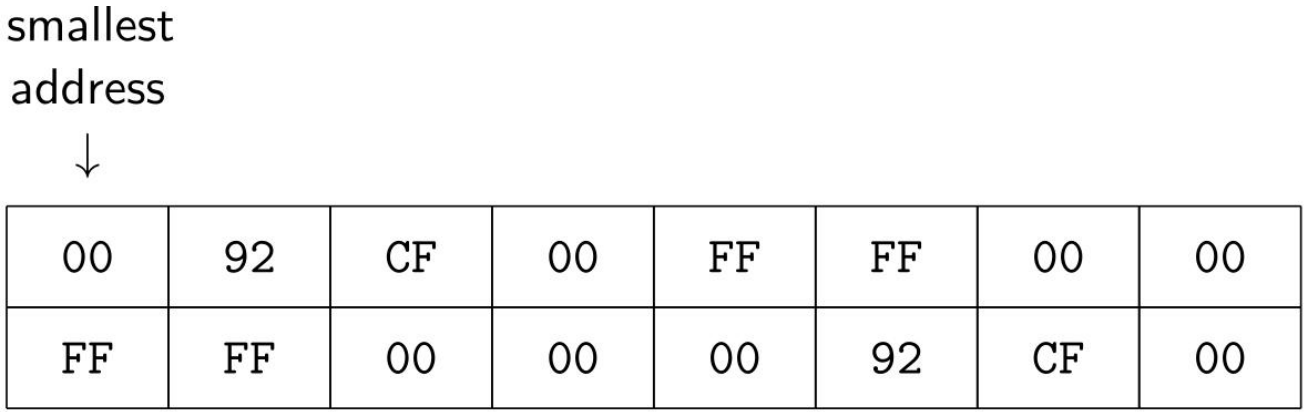
- Cosa succede se memorizziamo due interi di 4 byte in modo consecutivo in memoria?
- Prendiamo ad esempio `0x00cf9200` e `0x0000ffff`




- I byte di ogni singolo intero sono scambiati, ma non l'ordine degli interi

# Effetti della Little-Endianness

- Cosa succede se memorizziamo due interi di 4 byte, seguiti da un intero di 8 byte?
- Prendiamo ad esempio `0x00cf9200`, `0x0000ffff` e `0x00cf92000000ffff`



# Little-Endianness

- Apparentemente questa rappresentazione dei dati è una complicazione
- La CPU deve infatti “ribaltare” ogni volta i dati, o le componenti della PU devono lavorare a byte invertiti
- Cosa succede con le *conversioni di dati* (*cast*)?

- L’indirizzo di memoria non cambia se voglio accedere ad una sottoporzione del dato
- I processori Big-Endian devono invece calcolare uno spiazzamento corretto per accedere a sottoporzioni del dato.
- L’architettura x86 (e quindi lo z64) sono processori Litte-Endian

# Instruction Set

## Le istruzioni supportate dallo z64

# Le istruzioni dello z64

- Le seguenti convenzioni vengono utilizzate per rappresentare gli operandi delle istruzioni:
  - **B** — L'operando è un registro di uso generale, un indirizzo di memoria o un valore immediato. In caso di un indirizzo di memoria, qualsiasi combinazione delle modalità di indirizzamento è lecita. In caso di un immediato, la sua posizione dipende dalla possibile presenza dello spiazzamento e dalla sua dimensione.
  - **E** — L'operando è un registro di uso generale, o un indirizzo di memoria. In caso di un indirizzo di memoria, qualsiasi combinazione delle modalità di indirizzamento è lecita.
  - **G** — L'operando è un registro di uso generale.
  - **K** — L'operando è una costante numerica non segnata di valore fino a $2^{32} - 1$
  - **M** — L'operando è una locazione di memoria, codificata come uno spiazzamento a partire dal contenuto del registro RIP dopo l'esecuzione della fase di fetch
  - **ImmK** — L'operando è un dato immediato di K cifre binarie

# Classe 0: Controllo hardware

- Le istruzioni di controllo hardware consentono di modificare lo stato della CPU, oppure eseguono istruzioni particolari

| Tipo | Mnemonico | Operandi | O S Z P C | Descrizione |
|---|---|---|---|---|
| 1 | `hlt` | – | - - - - - | Mette la CPU in modalità di basso consumo energetico, finché non viene ricevuta l'interruzione successiva |
| 2 | `nop` | – | - - - - - | Nessuna operazione |
| 3 | `int` | `Imm8` | - - - - - | Chiama esplicitamente un gestore di interruzioni |

# Classe 1: Istruzioni di movimento dei dati

| Tipo | Mnemonico | Operandi | O | S | Z | P | C | Descrizione |
|---|---|---|---|---|---|---|---|---|
| 0 | `mov` | B, E | - | - | - | - | - | Fa una copia di `B` in `E` |
| 1 | `movsX` | E, G | - | - | - | - | - | Fa una copia di `E` in `G` con estensione del segno |
| 2 | `movzX` | E, G | - | - | - | - | - | Fa una copia di `E` in `G` con estensione dello zero |
| 3 | `lea` | E, G | - | - | - | - | - | Valuta la modalità di indirizzamento, salva il risultato in `G` |
| 4 | `push` | E | - | - | - | - | - | Copia il contenuto di `E` sulla cima dello stack |
| 5 | `pop` | E | - | - | - | - | - | Copia il contenuto della cima dello stack in `E` |
| 6 | `pushf` | – | - | - | - | - | - | Copia sulla cima dello stack il registro FLAGS |
| 7 | `popf` | – | - | - | - | - | - | Copia nel registro FLAGS il contenuto della cima dello stack |
| 8 | `movs` | – | - | - | - | - | - | Esegue una copia memoria-memoria |
| 9 | `stos` | – | - | - | - | - | - | Imposta una regione di memoria ad un dato valore |

# Estensioni del segno

| Istruzione | Tipo di conversione |
|---|---|
| `movsbw %al, %ax` | Estendi il segno da byte a word |
| `movsbl %al, %eax` | Estendi il segno da byte a longword |
| `movsbq %al, %rax` | Estendi il segno da byte a quadword |
| `movswl %ax, %eax` | Estendi il segno da word a longword |
| `movswq %ax, %rax` | Estendi il segno da word a quadword |
| `movslq %eax, %rax` | Estendi il segno da longword a quadword |

- L'istruzione movzX supporta le stesse combinazioni di suffissi
- I nomi dei registri virtuali devono essere coerenti con i suffissi

# Classe 2: Istruzioni logico/aritmetiche (1)

| Tipo | Mnemonico | Operandi | O | S | Z | P | C | Descrizione |
|---|---|---|---|---|---|---|---|---|
| 0 | add | B, E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Memorizza in E il risultato di E + B |
| 1 | sub | B, E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Memorizza in E il risultato di E - B |
| 2 | adc | B, E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Memorizza in D il risultato di E + B + CF |
| 3 | sbb | B, E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Memorizza in D il risultato di E - (B + neg(CF)) |
| 4 | cmp | B, E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Confronta i valori di B ed E calcolando E - B, il risultato viene poi scartato |
| 5 | test | B, E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Calcola l'and logico bit a bit di B ed E, il risultato viene poi scartato |
| 6 | neg | E | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Rimpiazza il valore di E con il suo complemento a 2 |
| 7 | and | B, E | 0 | ⇕ | ⇕ | ⇕ | 0 | Memorizza in E il risultato dell'and bit a bit tra B ed E |
| 8 | or | B, E | 0 | ⇕ | ⇕ | ⇕ | 0 | Memorizza in E il risultato dell'or bit a bit tra B ed E |

# Aritmetica a precisione arbitraria

- Con lo z64 è possibile effettuare operazioni aritmetiche (somme e sottrazioni) usando dati a 64 bit
- Può essere necessario effettuare operazioni con dati più grandi
- Le istruzioni `adc` e `sbb` permettono di realizzare programmi a *precisione arbitraria*

```
movq $operand_1_high, %rax
movq $operand_1_low, %rbx
movq $operand_2_high, %rcx
movq $operand_2_low, %rdx
addq %rbx, %rdx
adcq %rax, %rcx
```

# Classe 2: Istruzioni logico/aritmetiche (2)

| Tipo | Mnemonico | Operandi | O | S | Z | P | C | Descrizione |
|---|---|---|---|---|---|---|---|---|
| 9 | `xor` | B, E | 0 | ⇕ | ⇕ | ⇕ | 0 | Memorizza in `E` il risultato dello xor bit a bit tra `B` ed`E` |
| 10 | `not` | E | 0 | ⇕ | ⇕ | ⇕ | 0 | Rimpiazza il valore di `E` con il suo complemento a uno |
| 11 | `bt` | K, E | - | - | - | - | ⇕ | Imposta `CF` al valore del K-simo bit di `E` (bit testing) |
| 12 | `mul` | E | ⇕ | ? | ? | ? | ⇕ | Moltiplicazione senza segno (RDX:RAX $\leftarrow$ RAX $\cdot$ S) |
| 13 | `imul` | E | ⇕ | ? | ? | ? | ⇕ | Moltiplicazione con segno (RDX:RAX $\leftarrow$ RAX $\cdot$ S) |
| 14 | `div` | E | ? | ? | ? | ? | ? | Divisione senza segno di RDX:RAX per S, con RAX $\leftarrow$ quoziente e RDX $\leftarrow$ resto. |
| 15 | `div` | E | ? | ? | ? | ? | ? | Divisione senza segno di RDX:RAX per S, con RAX $\leftarrow$ quoziente e RDX $\leftarrow$ resto. |

# Classe 3: Istruzioni di rotazione e shift

| Tipo | Mnemonico | Operandi | O | S | Z | P | C | Descrizione |
|---|---|---|---|---|---|---|---|---|
| 0 | sal | K, G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Moltiplica per 2, K volte |
| 1 | sal | G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Moltiplica per 2, RCX volte |
| 0 | shl | K, G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Moltiplica per 2, K volte |
| 1 | shl | G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Moltiplica per 2, RCX volte |
| 2 | sar | K, G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Dividi (con segno) per 2, K volte |
| 3 | sar | G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Dividi (con segno) per 2, RCX volte |
| 4 | shr | K, G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Dividi (senza segno) per 2, K volte |
| 5 | shr | G | ⇕ | ⇕ | ⇕ | ⇕ | ⇕ | Dividi (senza segno) per 2, RCX volte |
| 6 | rcl | K, G | ⇕ | - | - | - | ⇕ | Ruota a sinistra, K volte |
| 7 | rcl | G | ⇕ | - | - | - | ⇕ | Ruota a sinistra, RCX volte |
| 8 | rcr | K, G | ⇕ | - | - | - | ⇕ | Ruota a destra, K volte |
| 9 | rcr | G | ⇕ | - | - | - | ⇕ | Ruota a destra, RCX volte |
| 10 | rol | K, G | ⇕ | - | - | - | ⇕ | Ruota a sinistra, K volte |
| 11 | rol | G | ⇕ | - | - | - | ⇕ | Ruota a sinistra, RCX volte |
| 12 | ror | K, G | ⇕ | - | - | - | ⇕ | Ruota a destra, K volte |
| 13 | ror | G | ⇕ | - | - | - | ⇕ | Ruota a destra, RCX volte |

# Classe 4: Manipolazione dei bit di FLAGS

| Tipo | Mnemonico | Operandi | O | S | Z | P | C | Descrizione |
|---|---|---|---|---|---|---|---|---|
| 0 | clc | – | - | - | - | - | 0 | Resetta CF |
| 1 | clp† | – | - | - | - | 0 | - | Resetta PF |
| 2 | clz† | – | - | - | 0 | - | - | Resetta ZF |
| 3 | cls† | – | - | 0 | - | - | - | Resetta SF |
| 4 | cli | – | - | - | - | - | - | Resetta IF |
| 5 | cld | – | - | - | - | - | - | Resetta DF |
| 6 | clo† | – | 0 | - | - | - | - | Resetta OF |
| 7 | stc | – | - | - | - | - | 1 | Imposta CF |
| 8 | stp† | – | - | - | - | 1 | - | Imposta PF |
| 9 | stz† | – | - | - | 1 | - | - | Imposta ZF |
| 10 | sts† | – | - | 1 | - | - | - | Imposta SF |
| 11 | sti | – | - | - | - | - | - | Imposta IF |
| 12 | std | – | - | - | - | - | - | Imposta DF |
| 13 | sto† | – | 1 | - | - | - | - | Imposta OF |

†: non esiste un'istruzione corrispondente nell'assembly x86

# Classe 5: Controllo del flusso di programma

| Tipo | Mnemonico | Operandi | O | S | Z | P | C | Descrizione |
|---|---|---|---|---|---|---|---|---|
| 0 | `jmp` | M | - | - | - | - | - | Esegue un salto relativo |
| 1 | `jmp` | *G | - | - | - | - | - | Esegui un salto assoluto |
| 2 | `call` | M | - | - | - | - | - | Esegue una chiamata a subroutine relativa |
| 3 | `call` | *G | - | - | - | - | - | Esegue una chiamata a subroutine assoluta |
| 4 | `ret` | – | - | - | - | - | - | Ritorna da una subroutine |
| 5 | `iret` | – | ⇳ | ⇳ | ⇳ | ⇳ | ⇳ | Ritorna dal gestore di una interruzione |

# Classe 6: Controllo condizionale del flusso

| Tipo | Mnemonico | Operandi | O S Z P C | Descrizione |
|---|---|---|---|---|
| 0 | jc | M | - - - - - | Salta a M se CF è impostato |
| 1 | jp | M | - - - - - | Salta a M se PF è impostato |
| 2 | jz | M | - - - - - | Salta a M se ZF è impostato |
| 3 | js | M | - - - - - | Salta a M se SF è impostato |
| 4 | jo | M | - - - - - | Salta a M se OF è impostato |
| 5 | jnc | M | - - - - - | Salta a M se CF non è impostato |
| 6 | jnp | M | - - - - - | Salta a M se PF non è impostato |
| 7 | jnz | M | - - - - - | Salta a M se ZF non è impostato |
| 8 | jns | M | - - - - - | Salta a M se SF non è impostato |
| 9 | jno | M | - - - - - | Salta a M se OF non è impostato |